Daniel Lucrédio

Uma Abordagem Orientada a Modelos para Reutilização de Software

USP – São Carlos – SP

Julho de 2009

Daniel Lucrédio

Uma Abordagem Orientada a Modelos para Reutilização de Software

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional.

Orientadora:
Profª Drª Renata Pontin de Mattos Fortes

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
UNIVERSIDADE DE SÃO PAULO

USP – São Carlos – SP

Julho de 2009

Agradecimentos

Agradeço à minha orientadora, Renata, que me acolheu e orientou durante todos estes anos. Além dos diversos papéis que lhe cabem oficialmente, sendo uma orientadora dedicada, interessada e competente, ela me ajudou muito dentro e fora do doutorado.

Obrigado também aos membros da banca examinadora, Alessandro Fabricio Garcia, Claudia Maria Lima Werner, Ivan Luiz Marques Ricarte e Rosana Teresinha Vaccare Braga, que com seus valiosos comentários durante a defesa ajudaram a enriquecer muito este trabalho.

Agradeço também às instituições que fizeram com que esta trajetória fosse possível. Estas incluem o ICMC-USP, por ter sido a minha casa durante estes anos. Também agradeço à FAPESP, CAPES e CNPq, que em diversos momentos me ajudaram financeiramente. Espero ter correspondido e continuar correspondendo à altura este investimento em mim realizado.

Agradeço ao grupo RiSE (*Reuse in Software Engineering*), e as suas instituições de apoio, UFPE e C.E.S.A.R. em Recife-PE. Tenho orgulho em ter participado do começo desta iniciativa, junto com Eduardo Santana de Almeida, Vinicius Cardoso Garcia e Alexandre Alvaro, sob a orientação de Silvio Meira. Se hoje o RiSE está entre os principais grupos de reutilização do mundo, é porque realmente compreendemos o verdadeiro significado da palavra "colaboração".

Durante o doutorado, passei seis meses na George Mason University, em Fairfax, VA, Estados Unidos, sob a orientação de Jon Whittle, onde aprendi muitas coisas, somente algumas delas contidas nesta tese de uma forma ou de outra. Foram também três meses em Redmond, WA, Estados Unidos, na Microsoft Research, sob a orientação de Ethan Jackson e Wolfram Schulte, em **outro** grupo RiSE, esse chamado *Research in Software Engineering*. Este estágio foi parte de um prêmio que recebi, o Latin American Fellowship Award 2008-2009, oferecido pela Microsoft Research. Posso seguramente dizer que essa experiência deixou marcas e aprendizados que foram além de todas as minhas expectativas, e é impossível não ficar grato pelo reconhecimento de meu trabalho. Agradeço também a Jaime Puente, Juliana Salles, e toda a equipe da Microsoft responsável por essa ótima iniciativa.

Obrigado à Aptor Software, por acreditar neste projeto, cedendo parte de seu tempo e seus funcionários para a realização de um dos estudos empíricos.

Agradeço também aos colegas e amigos que encontrei pelo caminho, pois foram muito

importantes durante essa trajetória: no ICMC, Débora, Daniel, Thiago, André, Silvana, Américo, Adalberto, Willian, David, Eduardo e Prazeres; em Recife-PE, Eduardo, Vinicius, Alexandre, Fred, Bart, Liana, Vanilson, Kellyton, e demais membros do RiSE; Em Fairfax, EUA, Marcos, Rodrigo, William, Aline e Cristiane; e finalmente Leo, Renan, Alexandre, Alexandro, Matko, e os outros estagiários da MSR.

Durante o doutorado, foi também importante o período de dois anos e meio trabalhando em tempo parcial na 3WT, empresa aqui de São Carlos. Não existe um vínculo oficial entre o que fiz na empresa e o que fiz na universidade, mas a vivência prática na indústria foi sem dúvida relevante para esta tese, e agradeço à empresa e aos amigos que nela fiz: Dugnani, Evandro, Cristiano, Escovar, Danilo, Vital, Ricardo, Rodrigo, Mateus, Renan, Cris, Maria e muitos outros com quem convivi neste período.

Agradeço também aos amigos Cassandra, Marcelo, Bianca, Amandinha, Ivan e Amanda, pela amizade e pelas noites de descontração.

Por fim, agradeço à minha amada esposa Alessandra e minha filha Julia, por todo o amor e carinho que me fazem lembrar que a vida não podia ser melhor. E também pelo apoio, força e compreensão nos (muitos) momentos difíceis. A todos interessados em iniciar uma jornada como a de um doutorado, recomendo fortemente a presença da família. Também agradeço a meus pais Horácio e Dalila, meus irmãos e cunhados, Rafael e Amanda, Maurício e Andréa, Fábio e Andréa, meus sogros Vivaldo e Ivani e meus sobrinhos, Gabriel, Leo, Dudu e Giovanna. E também a Guiza, minha companheira fiel durante grande parte do desenvolvimento e escrita da tese.

E obrigado DEUS, pela minha vida.

"Academic organizations in computer science seem to prefer to work on new theories. However, some of the most successful academic work is a fusion and formalization of successful techniques."

James M. Neighbors

Resumo

A reutilização de software busca aumentar a qualidade e produtividade no desenvolvimento de software, evitando a duplicação do esforço e reaproveitando o máximo possível das experiências de projetos passados. Apesar de simples, esta idéia não é facilmente colocada em prática, principalmente de maneira sistemática e controlada. Técnicas de engenharia de domínio e linhas de produtos de software buscam facilitar esta tarefa, porém ainda existem outros fatores que dificultam a adoção da prática da reutilização. Entre estes, destacam-se os problemas inerentes ao desenvolvimento de software da maneira como é conduzido atualmente, baseado em código-fonte. Estes problemas têm suas origens na crescente demanda por software cada vez mais complexo e afetam negativamente a capacidade de reutilizar software. O desenvolvimento orientado a modelos surge como uma alternativa atraente neste cenário, elevando a importância de modelos dentro do ciclo de vida do software, incorporando-os como parte integrante do produto final por meio de técnicas de modelagem e geração de código. Com isto, parte da complexidade do software fica escondida dentro dos geradores, protegendo os desenvolvedores, reduzindo a incidência de erros, aumentando a produtividade, qualidade, interoperabilidade e manutenibilidade dos artefatos produzidos. Nesta dissertação defende-se a tese de que o desenvolvimento orientado a modelos pode efetivamente aumentar e/ou melhorar a reutilização de software, e que para isso ele deve ser tratado de forma consistente dentro de um processo de engenharia de domínio. Em particular, ele deve reconhecer que não é possível automatizar o domínio todo. Deve identificar os múltiplos subdomínios com diferentes graus de automação e gerenciar a sua integração desde a análise até a implementação. Para demonstrar esta tese, é apresentada uma abordagem orientada a modelos para reutilização de software, com atividades que guiam o desenvolvedor durante a análise, projeto e implementação do domínio. São também apresentados os resultados de uma avaliação envolvendo três estudos empíricos, realizados em ambiente acadêmico e industrial, que buscou determinar a viabilidade da abordagem e os benefícios que podem ser alcançados com a combinação de técnicas do desenvolvimento orientado a modelos e da reutilização de software. Os resultados mostram que a abordagem pode trazer diferentes benefícios para organizações de software, incluindo aumento da quantidade e qualidade da reutilização, e reduzindo a complexidade de desenvolvimento e configuração de produtos.

Palavras-chave: Reutilização de software, Desenvolvimento Orientado a Modelos, Engenharia de Domínio, Linhas de Produtos de Software, Linguagens Específicas de Domínio, Programação Generativa.

Abstract

Software reuse aims at increasing quality and productivity in software development, avoiding effort duplication and reusing all that is possible from past experiences. Although it is a simple idea, it is not easy to put reuse in practice, especially in a systematic and controlled way. Domain engineering and software product lines techniques try to make this task easier, but there are many other factors that make the reuse adoption difficult. Among these factors are problems that are inherent to software development in the way it is conducted today, based on source code. These problems arise from the growing demand for increasingly complex software, negatively affecting the ability to reuse. Model-driven development is an attractive alternative in this scenario, leveraging the importance of models in the software life cycle, incorporating them as part of the final product through modeling and code generation techniques. As a result, part of the software complexity becomes hidden inside the generators, shielding the developers, reducing errors, increasing the productivity, quality, interoperability and maintainability of the produced assets. This dissertation presents the thesis that model-driven development can effectively increase and/or improve software reuse, and that to achieve this goal it must be treated in a consistent way inside a domain engineering process. Particularly, it must acknowledge that it is not possible to fully automate a domain. It must identify the multiple subdomains with different automation degrees and manage their integration from the analysis to the implementation. To demonstrate this thesis, a model-driven software reuse approach is presented, with activities that guide the developer during domain analysis, design and implementation. The results of an evaluation involving three empirical studies are also presented. The studies were performed in both academic and industrial environments, and aimed at determining the viability of the approach and the benefits that can be achieved with the combination of model-driven development and software reuse techniques. The results showed that the approach can bring different benefits to software organizations, such as software reuse quantity and quality improvements, and complexity reduction in product development and configuration tasks.

Keywords: Software Reuse, Model-Driven Development, Domain Engineering, Software Product Lines, Domain-Specific Languages, Generative Programming.

Lista de Figuras

1 Modelo de maturidade em reutilização 35

2 Ilustração do processo de criação de software no desenvolvimento orientado a modelos 41

3 Principais elementos do MDD 44

4 Arquitetura clássica de metamodelagem 45

5 Funcionamento do MDR 49

6 Comparação entre MDR e EMF 50

7 Modelo de maturidade em MDD 53

8 Geração de código baseada em *templates* 66

9 Sugestão de modelo de processo para utilização da abordagem 75

10 Abrangência da abordagem, em relação aos modelos de maturidade em reutilização e MDD 81

11 Modelo de *features* do domínio web de autoria de conteúdo 93

12 Candidatos a subdomínio do domínio web de autoria de conteúdo 102

13 Possível evolução dos níveis de decisão para o domínio de aplicações de autoria de conteúdo para Web 110

14 Modelo de *features* do domínio web de autoria de conteúdo 116

15 Uso do padrão *Abstract Factory* para *features* alternativas 126

16 Uso do padrão *Prototype* para *features* alternativas 127

17 Uso do padrão *Template method* para *features* alternativas 128

18 Uso do padrão *Chain of Responsibility* para *or features* 129

19 Uso do padrão *Decorator* para *or features* 130

20 Padrão *visitante* sendo aplicado à implementação de variabilidade baseada em *features* 131

21 Abordagem *template* sendo aplicada à geração de código baseada em *features* . 132

22 Padrão camada fina de dados 133

23 "*Merging*" de geradores envolvendo modelos estruturais e comportamentais . . 136

24 Estratégia de caracterização de subdomínios 147

25 Modelo de *features* do domínio web de autoria de conteúdo 148

26 Definição do metamodelo da DSL de autoria Web 151

27 Manutenção da consistência da implementação de referência 161

28 Comparação entre as métricas de reutilização para o estudo do domínio de autoria de conteúdo para a web 189

29 Comparação entre as métricas indiretas de reusabilidade para o estudo do domínio de autoria de conteúdo para a web 189

30 Distribuições de instabilidade de módulo nos diferentes tipos de artefatos produzidos com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web 190

31 Distribuições de complexidade ciclomática nos diferentes tipos de artefatos produzidos com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web 190

32 Distribuições do índice de manutenibilidade nos diferentes tipos de artefatos produzidos com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web 191

33 Distribuições do número de atributos e do número de relacionamentos nos modelos utilizados com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web 191

34 Comparação entre as métricas de reutilização para o estudo do domínio de aplicações distribuídas baseadas em computação em nuvem 193

35 Comparação entre as métricas indiretas de reusabilidade para o estudo do domínio de aplicações distribuídas baseadas em computação em nuvem 194

36 Distribuições de instabilidade de módulo nos diferentes tipos de artefatos produzidos com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem 195

37 Distribuições de complexidade ciclomática nos diferentes tipos de artefatos produzidos com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem 195

38 Distribuições do índice de manutenibilidade nos diferentes tipos de artefatos produzidos com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem 196

39 Distribuições de números de atributos e relacionamentos nos modelos utilizados com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem 196

40 Comparação entre as métricas de reutilização para o estudo do domínio de eventos científicos 197

41 Comparação entre as métricas de reusabilidade dos estudos dos domínios de eventos científicos (EC), Web e Computação em Nuvem (CN) 198

42 Efeito da abordagem na reusabilidade dos artefatos produzidos 202

43 Uma possível hipótese alternativa elaborada a partir dos resultados da avaliação 203

44 Processo de refinamentos sucessivos na abordagem Draco 210

45 Situação clássica da análise de sistemas 257

Lista de Quadros

1 Identificação inicial das *features* de duas aplicações do domínio de autoria de conteúdo para Web 91

2 Compilação das *features* de quatro aplicações do domínio de autoria de conteúdo para Web 92

3 Exemplo de caso de uso do domínio web 94

4 Exemplo de caso de mudança para a *feature* opcional de busca 94

5 Exemplo de caso de mudança para a *feature* alternativa de busca avançada . . . 95

6 Documentação do subdomínio de navegação 106

7 Documentação do nível de confiança dos subdomínios identificados 107

8 Resumo das atividades para análise de domínio orientada a modelos 112

9 Descrição dos produtos de trabalho da fase de análise de domínio 113

10 Resumo das atividades para projeto de domínio orientado a modelos 141

11 Descrição dos produtos de trabalho da fase de projeto de domínio 142

12 Resumo das atividades para implementação do domínio orientada a modelos . . 167

13 Descrição dos produtos de trabalho da fase de implementação do domínio . . . 168

14 Dados sobre o estudo envolvendo o domínio de autoria de conteúdo para Web . 183

15 Dados sobre o estudo envolvendo o domínio de aplicações distribuídas baseadas em computação em nuvem 185

16 Dados sobre o estudo envolvendo o domínio de eventos científicos 186

17 Resumo das principais técnicas relacionadas aos conceitos básicos da reutilização de software 270

Lista de Acrônimos

ADD *Attribute-Driven Design* / Projeto Orientado a Atributos

AOP *Aspect-Oriented Programming* / Programação Orientada a Aspectos

AST *Abstract Syntax Tree* / Árvore de Sintaxe Abstrata

ATL *Atlas Transformation Language* / Linguagem de Transformação Atlas

BNF *Backus-Naur Form* / Forma de Backus-Naur

CIM *Computation Independent Model* / Modelo Independente de Computação

CVS *Concurrent Versions System* / Sistema de Versões Concorrentes

DAO *Data Access Object* / Objeto de Acesso a Dados

DBC Desenvolvimento Baseado em Componentes

DBML *DataBase Modeling Language* / Linguagem de Modelagem de Banco de Dados

DSL *Domain-Specific Language* / Linguagem Específica de Domínio

ED Engenharia de Domínio

EJB *Enterprise JavaBeans*

EMF *Eclipse Modeling Framework* / *Framework* de Modelagem do Eclipse

FBU *Feature Binding Unit* / Unidade de Agrupamento de *Features*

GME *Generic Modeling Environment* / Ambiente de Modelagem Genérico

GMF *Graphical Modeling Framework* / *Framework* de Modelagem Gráfica

GMT *Generative Modeling Tools* / Ferramentas de Modelagem Generativas

GPL *General Purpose Language* / Linguagem de Propósito Geral

GQM *Goal-Question Metric* / Métrica Objetivo-Questão

GUI *Graphical User Interface* / Interface Gráfica com o Usuário

IDE *Integrated Development Environment* / Ambiente de Desenvolvimento Integrado

JET *Java Emitter Templates* / Modelos Emissores de Java

JMI *Java Metadata Interface* / Interface para Metadados em Java

JSP *Java Server Pages* / Páginas Servidoras Java

LOC *Lines Of Code* / Linhas De Código

MDA *Model-Driven Architecture* / Arquitetura Orientada a Modelos

MDD *Model-Driven Development* / Desenvolvimento Orientado a Modelos

MDE *Model-Driven Engineering* / Engenharia Orientada a Modelos

MDR *MetaData Repository* / Repositório de MetaDados

MDSD *Model-Driven Software Development* / Desenvolvimento de Software Orientado a Modelos

MD* Acrônimo que abrange todas as abordagens de desenvolvimento de software orientadas a modelos

MIC *Model Integrated Computing* / Computação Integrada por Modelos

MOF *Meta-Object Facility* / Infraestrutura de MetaObjetos

MTF *Model Transformation Framework* / *Framework* de Transformação de Modelos

MVC *Model-View-Controller* / Modelo-Visão-Controlador

oAW *openArchitectureWare* - conjunto de ferramentas para desenvolvimento orientado a modelos

OMG *Object Management Group* / Grupo de Gerenciamento de Objetos

OTAN Organização do Tratado do Atlântico Norte

PHP *PHP: Hypertext Preprocessor* / PHP: Pré-processador de Hipertexto[1]

PNRP *Peer Name Resolution Protocol* / Protocolo de Resolução de Nomes de Pares

[1] acrônimo recursivo

POSA *Pattern-Oriented Software Architecture* / Arquitetura de Software Orientada a Padrões

PIM *Platform Independent Model* / Modelo Independente de Plataforma

PSM *Platform Specific Model* / Modelo Específico de Plataforma

PuLSE *ProdUct-Line Software Engineering* / Engenharia de Software de Linha de Produtos

QVT *Queries/Views/Transformations* / Consultas/Visões/Transformações

RAS *Reusable Asset Specifications* / Especificações de Artefatos Reutilizáveis

RMM *Reuse Maturity Model* / Modelo de Maturidade em Reutilização

RPC *Reuse Capability Model* / Modelo de Capacitação em Reutilização

RRM *Reuse Reference Model* / Modelo de Referência em Reutilização

RUP *Rational Unified Process* / Processo Unificado da *Rational*

SAAM *Software Architecture Analysis Method* / Método de Avaliação de Arquitetura de Software

SCV *Scope, Commonality, and Variability* / Escopo, Comunalidade e Variabilidade

SPC *Software Productivity Consortium* / Consórcio de Produtividade de Software

SVN Acrônimo normalmente utilizado como referência ao sistema *SubVersioN* para controle de versões

UML *Unified Modeling Language* / Linguagem de Modelagem Unificada

XMI *XML Metadata Interchange* / Intercâmbio de Metadados em XML

Sumário

1 Introdução **19**

1.1 A tese 23

1.2 Definição do escopo 25

1.3 Estrutura da dissertação 26

2 Conceitos envolvidos **29**

2.1 Reutilização de software 29

2.2 Desenvolvimento orientado a modelos 36

2.3 Considerações finais 55

3 Reutilização de software e desenvolvimento orientado a modelos **57**

3.1 Análise SCV 57

3.2 Implementação da variabilidade 62

3.3 Considerações finais 67

4 Visão geral da abordagem **69**

4.1 Objetivo da abordagem 69

4.2 Definição da abordagem 71

4.3 Estrutura da abordagem 72

4.4 Modelo de processo 74

4.5 Abrangência da abordagem 81

4.6 Considerações finais 84

5 Análise de domínio orientada a modelos **85**

5.1 Objetivos da análise de domínio 86
5.2 Atividades da análise de domínio 87
5.3 Considerações finais 112

6 Projeto de domínio orientado a modelos 115

6.1 Objetivos do projeto de domínio 118
6.2 Atividades do projeto de domínio 119
6.3 Considerações finais 140

7 Implementação de domínio orientada a modelos 143

7.1 Objetivos da implementação do domínio 144
7.2 Atividades da implementação do domínio 146
7.3 Considerações finais 166

8 Avaliação 169

8.1 Definição 169
8.2 Descrição dos projetos utilizados nos estudos empíricos e sua execução 181
8.3 Coleta dos dados 186
8.4 Resultados e discussão 188
8.5 Análise das hipóteses e conclusões 201
8.6 Ameaças à validade 202
8.7 Considerações finais 204

9 Trabalhos relacionados 207

9.1 Abordagens orientadas a modelos para reutilização de software 207
9.2 Trabalhos relacionados com o uso de métricas para MDD e reutilização 222
9.3 Considerações finais 226

10 Conclusões 227

10.1 Principais contribuições 227
10.2 Lições aprendidas 228
10.3 Limitações e Trabalhos futuros 229
10.4 Considerações finais 232

Referências **235**

Apêndice A – Técnicas para reutilização de software **253**

Apêndice B – Relação entre a abordagem e modelos de maturidade **271**

Apêndice C – Reprodução na íntegra da entrevista referente ao estudo empírico envolvendo o domínio de eventos científicos **283**

1 Introdução

Há duas décadas, Krueger (1992) colocou a reutilização de software como uma preocupação constante para organizações que buscam aumento na qualidade e produtividade em seu processo de desenvolvimento. A qualidade pode ser melhorada por meio da reutilização de experiências comprovadas, incluindo artefatos em diferentes níveis de abstração, produtos e processos. Neste sentido, também ocorre um aumento na produtividade, uma vez que essas experiências são reaproveitadas, evitando-se a necessidade de construí-las novamente (BASILI; BRIAND; MELO, 1996).

No entanto, a idéia de reutilizar ao invés de construir é ainda mais antiga, remetendo às origens da própria engenharia de software. Em 1968, na conferência da OTAN que é considerada como o berço da engenharia de software[1], McIlroy (1968) apresentava idéias referentes à reutilização de componentes de software produzidos em massa e os possíveis benefícios que tal abordagem traria à chamada "*crise do software*"[2].

Desde então, a pesquisa na área evoluiu, produzindo uma diversidade de trabalhos envolvendo reutilização de software. Os esforços que podem ser encontrados na literatura incluem as idéias de engenharia de domínio (NEIGHBORS, 1980; STARS, 1993; SIMOS et al., 1996; JACOBSON; GRISS; JONSSON, 1997; GRISS; FAVARO; D'ALESSANDRO, 1998; KANG et al., 1998), desenvolvimento baseado em componentes (DBC) (SZYPERSKI, 1999; SAMETINGER, 1997), e mais recentemente, as idéias de linhas de produto (CLEMENTS; NORTHROP, 2002). Existe também uma vasta literatura na área de repositórios e busca de artefatos (LUCRÉDIO; ALMEIDA; PRADO, 2004). Fora da academia, diferentes relatos sobre fatores e práticas de sucesso podem ser encontrados, em empresas como Motorola (JOOS, 1994) e Hewlett-Packard (GRISS, 1995).

Porém, mesmo após décadas de pesquisa, ainda não é possível afirmar que existe uma abordagem voltada para reutilização de software que seja amplamente utilizada e que permita

[1] O termo **Engenharia de Software** foi criado nesta conferência como uma provocação, para ressaltar as necessidades de se empregar, no desenvolvimento de software, os conceitos já consagrados em outras engenharias (NAUR; RANDELL, 1968).

[2] Famoso termo que também se originou nesta mesma conferência, referente ao problema de se construir sistemas de software grandes e confiáveis de uma maneira controlável e eficiente.

que uma organização alcance os benefícios de qualidade e produtividade almejados por McIlroy e muitos outros desde então. Um estudo recente (ALMEIDA, 2007) identificou que a maioria das abordagens originadas na academia é focada em partes isoladas do problema, sem considerar a relação existente entre elas, e que os casos de sucesso relatados por empresas se devem a fatores muito específicos, não sendo genéricos o bastante para serem aplicáveis fora de seu contexto original.

Deve ficar claro que reutilização de software não envolve somente fatores técnicos, como a utilização de métodos, ferramentas ou métricas. Envolve também fatores não-técnicos, tais como educação, treinamento, motivação, incentivos e comprometimento organizacional, além de um bom planejamento e o apoio gerencial (ALMEIDA et al., 2004).

Dentre os fatores técnicos, destaca-se o papel do processo de software, que ajuda a incluir no desenvolvimento uma maior capacidade de se lidar com as restrições de tempo e esforço, tornando-o mais parecido com engenharia e menos com arte (ROMBACH, 2000). Um processo que possa ser observado, controlado e constantemente melhorado pode contribuir para que os benefícios da reutilização sejam alcançados, por meio de práticas eficientes de trabalho disseminadas facilmente entre os indivíduos de uma organização, sem depender de talentos individuais.

A necessidade de um processo sistemático de reutilização

Em termos de reutilização de software, faz-se necessário um método flexível, que possa ser adaptado para diferentes situações, que possua diretivas e suporte suficientes, e que não seja muito vago, caso contrário, o mesmo não irá atender às necessidades de variadas situações da indústria sem uma forte interpretação e suporte adicionais (BAYER et al., 1999).

Essa opinião é compartilhada por muitos autores da área de reutilização, podendo ser encontrada em relatórios de empresas (ENDRES, 1993; BAUER, 1993; GRISS, 1994, 1995; JOOS, 1994), "*pesquisas informais*" (FRAKES; ISODA, 1994) e estudos empíricos (RINE, 1997; MORISIO; EZRAN; TULLY, 2002; ROTHENBERGER et al., 2003). Todos esses trabalhos mostraram que adotar um processo pode ser uma maneira efetiva de se alcançar os benefícios da reutilização de software.

Porém, os processos existentes atualmente apresentam algumas falhas e falta de detalhes em atividades importantes, tais como o desenvolvimento **para reutilização** e o desenvolvimento **com reutilização**, além de ênfase em atividades específicas (análise, projeto ou implementação), e não no processo todo. Mesmo as recentes idéias de linhas de produto de software ainda não produziram consenso com relação a quais atividades (incluindo entradas,

saídas e artefatos produzidos) e requisitos um processo efetivo de reutilização deve possuir (ALMEIDA et al., 2005). Este fato também pode ser verificado no cenário brasileiro, onde um estudo de levantamento realizado em empresas (LUCRÉDIO et al., 2008) verificou que a maioria das organizações investigadas (cerca de 65%) não se preocupa com reutilização em seus processos de software.

Neste ponto chega-se a uma primeira motivação importante: faz-se necessário um processo sistemático e completo de reutilização, que complemente as idéias para resolver partes mais específicas do problema. Neste sentido, a combinação de conceitos e técnicas existentes em uma forma nova e inexplorada pode oferecer contribuições tanto para o mundo acadêmico como para a indústria. Esta opinião já foi expressa por Jim Neighbors, um dos pioneiros em reutilização de software, décadas atrás: *"organizações acadêmicas em ciência da computação parecem preferir o trabalho em novas teorias. Porém, alguns dos mais bem-sucedidos trabalhos acadêmicos são uma fusão e formalização de técnicas de sucesso."* (NEIGHBORS, 1989).

Os problemas com o desenvolvimento de software como realizado atualmente

Talvez a maior mudança que se pode testemunhar desde o nascimento da ciência da computação foi o grau com o qual computadores se tornaram essenciais em nossas vidas. Apenas imagine como sua rotina diária seria afetada se você não tivesse acesso a nenhum tipo de dispositivo computacional, e você poderá perceber o quão incrivelmente pervasiva esta tecnologia se tornou.

Neste sentido, o software – a alma de um computador, responsável pela operação destas máquinas vitais – precisa sobreviver em tal cenário de mudanças drásticas e constantes. A história mostrou que cientistas e profissionais da área da computação, especialmente aqueles envolvidos com desenvolvimento de software, são especialistas na arte da mudança. Novas linguagens de programação e ambientes de desenvolvimento integrados sempre buscaram seguir os avanços do hardware, e foram razoavelmente bem sucedidos até então.

Mas chegou-se a um ponto onde o software deve ser não apenas confiável, mas também operar em ambientes computacionais embarcados e distribuídos, comunicar-se utilizando uma grande variedade de paradigmas de comunicação, e se adaptar dinamicamente a mudanças nos ambientes operacionais (FRANCE; RUMPE, 2007). Isto se deve não apenas às tecnologias utilizadas, que se tornaram significativamente mais complexas nos últimos anos, mas também porque estas tecnologias mudam mais rapidamente do que os negócios para os quais as mesmas procuram dar suporte (UHL, 2003).

O fato é: software se tornou um artefato muito complexo. Ainda não tão complexo

ao ponto de não sermos mais capazes de construí-lo, mas o esforço necessário para tanto, utilizando tecnologias centradas em código-fonte, é muito grande (KLEPPE; WARMER; BAST, 2003; FRANCE; RUMPE, 2007).

Nas palavras de France e Rumpe (2007), este esforço chega a ser hercúleo: *"construir sistemas de software complexos de forma manual pode ser equiparado a construir pirâmides no antigo Egito. Nós nos maravilhamos com tais implementações de software de forma bastante parecida com a qual arqueólogos se maravilham com as pirâmides: essa admiração é principalmente baseada na apreciação do esforço requerido para se lidar com as significativas complexidades acidentais que surgem do uso de tecnologias inadequadas"*.

A proposta do MDD (*Model-Driven Development* ou Desenvolvimento Orientado a Modelos) é justamente resolver esses problemas, enfatizando a importância de modelos no processo de desenvolvimento de software (MELLOR; CLARK; FUTAGAMI, 2003). No MDD, os modelos não são "*apenas papel*" que auxiliam nas tarefas de desenvolvimento. Ao invés disso, eles são parte constituinte do software, assim como o código. A proposta é reduzir a distância semântica entre o domínio do problema e o domínio da implementação/solução, através de modelos de alto nível que protegem os desenvolvedores de software das complexidades da plataforma de implementação (FRANCE; RUMPE, 2007), e transformações que geram artefatos de implementação automaticamente, de forma a refletir a solução expressa nos modelos (SCHMIDT, 2006).

O MDD pode levar a consideráveis benefícios em termos de ganhos de produtividade e qualidade. Por exemplo, a Nokia relata conseguir desenvolver novos telefones celulares até dez vezes mais rápido e a Lucent reporta ganhos de produtividade de três a dez vezes, dependendo do produto (TOLVANEN, 2004). Wile (2004) cita uma redução de 50 para 1, em termos de linhas de especificação/código, que pode ser alcançada através de técnicas orientadas a modelos.

Conclui-se que o MDD não pode ser ignorado como uma forma efetiva de se melhorar as atuais práticas de desenvolvimento de software. Apesar das dificuldades técnicas e complexidade adicional para se construir o suporte necessário para o desenvolvimento orientado a modelos, no final pode ser vantajoso gastar esforços extras em busca de suas promessas.

Reutilização de software orientada a modelos

Os problemas levantados na seção anterior se tornam ainda mais críticos quando se fala em reutilização de software, uma vez que o princípio de se construir uma vez para reutilizar várias vezes requer que o software reutilizável – código-fonte e outros artefatos – seja facilmente adaptável a diferentes contextos, plataformas e ambientes. Isto significa que hoje um artefato

precisa ser mais reutilizável do que antes, porque não apenas ele precisa ser reutilizável em uma aplicação diferente na mesma plataforma e ambiente, mas também em uma aplicação diferente que roda em plataformas e ambientes diferentes.

Assim, a reutilização de software é uma das áreas que pode mais se beneficiar com o MDD. Os principais pesquisadores da área, como Neighbors (1980), Krueger (1992), Griss (1995), Frakes e Isoda (1994), Jacobson, Griss e Jonsson (1997), defendem a idéia de que os verdadeiros benefícios da reutilização de software somente podem ser atingidos quando artefatos de alto nível são reutilizados, além do código-fonte. Com o MDD, onde modelos são artefatos de primeira classe, isto é exatamente o que acontece.

Mas apesar de parecerem uma combinação natural (OMG, 2003), reutilização de software e desenvolvimento orientado a modelos são áreas bastante distintas. Há trabalhos que tentam combinar estes dois paradigmas, mas a extensão com a qual MDD é utilizado para incrementar reutilização – ou a reutilização é utilizada para incrementar o MDD – ainda não é suficiente. Existem abordagens de reutilização que utilizam MDD para automatizar partes do processo, mas elas ignoram outras partes importantes, como o projeto arquitetural visando o MDD, por exemplo. Da mesma forma, existem abordagens orientadas a modelos que focam em reutilização, mas geralmente elas enfatizam a implementação e o suporte a uma linguagem. O que se constata é que o mesmo que acontece com processos de reutilização em geral se repete com relação a processos de reutilização orientados a modelos - há uma necessidade de processos sistemáticos para MDD.

Mais especificamente, existe a questão sobre exatamente onde o MDD pode ser aplicado para automatizar a implementação de um domínio. Na prática, sabe-se que é muito difícil obter automação completa, isto é, ser capaz de completamente gerar uma aplicação sem a necessidade de modificá-la e/ou completá-la com código manualmente escrito. Porém, existem áreas de um domínio – ou subdomínios – onde o MDD pode efetivamente elevar a produtividade e reutilização. Identificar quais são esses subdomínios, e gerenciar seu ciclo de vida de forma que a implementação final possa integrar tanto o código que é gerado como o código não-gerado – que é escrito manualmente – de forma adequada, é o foco principal desta tese.

1.1 A tese

A tese está contextualizada na premissa da literatura de que a combinação entre o desenvolvimento orientado a modelos e reutilização de software em um processo sistemático pode elevar e/ou melhorar os níveis de reutilização alcançados por uma organização de software,

quando comparado com um processo não orientado a modelos. Para isso, a preocupação com o MDD deve estar presente em todas as etapas do processo, incluindo a análise, o projeto e a implementação.

Neste contexto, o foco desta tese está na identificação de como aplicar o MDD durante a engenharia de domínio, considerando-se que não é possível automatizá-lo completamente. A pergunta a ser respondida é: **durante o ciclo de vida da engenharia de domínio, como identificar e gerenciar os subdomínios automatizáveis e produzir uma arquitetura e implementação finais onde o potencial de automação destes subdomínios possa ser explorado e integrado com os demais subdomínios?**

A abordagem está centrada no conceito de subdomínios, e envolve a exploração dos seguintes pontos referentes à união entre MDD e reutilização de software:

- Como identificar, em um domínio, subdomínios que possam ser automatizados por meio de técnicas do MDD?
- Como lidar com o fato de que, para a identificação do potencial de automação de um subdomínio, pode ser necessário um esforço investigativo que envolve tempo e custos para uma organização?
- Como gerenciar os diferentes subdomínios e diferentes níveis de automação, considerando-se que a arquitetura e os componentes produzidos para o domínio devem ser capazes de promover a integração entre eles?
- Como gerenciar a variabilidade no domínio, considerando-se que o resultado da engenharia de domínio inclui não somente uma arquitetura e componentes reutilizáveis, mas também artefatos específicos do MDD, incluindo linguagens, transformações e geradores de código?
- Como gerenciar a existência de código gerado com código não-gerado, nos diferentes níveis de abstração em que são realizadas as atividades de análise, projeto e implementação?
- Em quais aspectos o MDD pode influenciar na reutilização de software? O MDD promove aumento e/ou melhoria na reutilização? Quais são os custos/benefícios de utilização do MDD, em um processo de engenharia de domínio?

Para validar esta tese, foi definida uma abordagem sistemática para reutilização de software, utilizando técnicas do desenvolvimento orientado a modelos, e que cobre

as etapas de análise, projeto e implementação de um domínio reutilizável. Foram também realizados estudos empíricos visando avaliar a abordagem definida e determinar o impacto do MDD na reutilização de software.

1.2 Definição do escopo

É importante ressaltar que o foco deste trabalho não foi pesquisar cada uma das linhas de pesquisa – reutilização e MDD – de forma individual e em profundidade. Isso seria um trabalho extremamente complexo que excederia o tempo estipulado para uma pesquisa em nível de doutorado. Ao invés disso, buscou-se combinar as técnicas já estabelecidas em um processo novo, de maneira a agregar os benefícios de ambas abordagens.

Além disso, foi priorizada a definição de uma abordagem prática, sem lacunas e usável, em vez de uma cobertura demasiadamente extensa do ciclo de vida do software. Esta não apenas foi uma das motivações desta pesquisa, mas também foi importante para que a avaliação final, que consistiu em colocar a abordagem em prática em cenários reais, pudesse ser realizada sem muitas dificuldades. Caso a abordagem apresentasse pontos indefinidos, ela não poderia ser utilizada nos estudos empíricos sem esforços adicionais.

Com base nestes critérios, um primeiro ponto a ser ressaltado é a cobertura do processo de reutilização. Trabalhou-se com o foco principal no desenvolvimento **para** a reutilização, pois sem este o desenvolvimento **com** reutilização não faz sentido. Foram definidas as atividades para construir os artefatos reutilizáveis (modelos, transformadores, geradores, etc.) de forma clara, podendo ser seguidas passo-a-passo em uma organização de software. O desenvolvimento **com** reutilização de uma maneira sistemática não foi abordado.

Com relação aos aspectos relacionados ao desenvolvimento orientado a modelos, o foco não está em definir um processo completo para MDD, e sim em **estender** um processo voltado à reutilização com os conceitos do MDD. Assim, foram definidas atividades para construção de modelos, transformadores e geradores a serem reutilizados no desenvolvimento de software. Está fora do escopo, portanto, tratar de atividades como manutenção e evolução destes artefatos.

Atividades de testes não são explicitamente tratadas nesta abordagem. Apesar de existirem vários aspectos a serem explorados com relação ao planejamento e realização de testes num contexto de reutilização e MDD, optou-se por não entrar neste mérito por uma questão de foco de pesquisa.

Outro ponto que dá margem a muitas possibilidades é o ambiente de suporte ao MDD. Ferramentas são parte essencial do MDD, e existem diversas opções para o desenvolvimento

orientado a modelos. Nesta pesquisa, trabalhou-se com ferramentas já existentes, tais como aquelas apresentadas na Seção 2.2.2, em vez de contruir o ambiente ou ferramentas a partir do zero. Isto foi possível devido ao atual cenário de MDD, que já conta com diversas opções concretas e que são efetivamente utilizadas na prática.

Em resumo, está **fora do escopo** desta pesquisa:

- A definição detalhada das atividades do desenvolvimento **com** reutilização;
- Atividades de manutenção dos artefatos, tais como aquelas relacionadas ao gerenciamento de configuração, por exemplo;
- Atividades para planejamento e execução de testes; e
- Desenvolvimento de ferramentas para o desenvolvimento orientado a modelos, tais como ferramentas para transformação ida-e-volta, ou ferramentas para validação de modelos e de transformações.

Mais detalhes sobre o escopo da pesquisa, em termos das práticas que estão incluídas e excluídas da abordagem definida nesta tese, encontram-se no Capítulo 4.

1.3 Estrutura da dissertação

Neste primeiro capítulo apresentou-se a motivação e objetivos da pesquisa. O restante da dissertação está estruturado da seguinte maneira:

- No Capítulo 2 discutem-se os fundamentos sobre a reutilização de software e o desenvolvimento orientado a modelos, incluindo os principais conceitos, técnicas e abordagens existentes;
- No Capítulo 3 são discutidos alguns pontos referentes à intersecção entre os conceitos da reutilização de software e do desenvolvimento orientado a modelos;
- No Capítulo 4 é apresentada uma visão geral da abordagem orientada a modelos para reutilização de software;
- Nos Capítulos 5, 6 e 7 são apresentadas as três fases da abordagem: análise, projeto e implementação do domínio;
- No Capítulo 8 são apresentados resultados da avaliação da abordagem;

- No Capítulo 9 são apresentados os trabalhos relacionados; e
- No Capítulo 10 são apresentadas as considerações finais, contribuições e trabalhos futuros.

2 *Conceitos envolvidos*

Esta tese envolveu duas abrangentes linhas de pesquisa: reutilização de software e desenvolvimento orientado a modelos. Com o objetivo de esclarecer melhor cada linha e seus pontos relevantes a esta tese, neste capítulo são apresentados os principais conceitos e técnicas relacionados a essas duas linhas.

2.1 Reutilização de software

A reutilização de software já foi considerada como uma "bala de prata" para resolver os problemas da crise do software. A realidade, porém, mostrou que forjar tal bala é muito mais difícil do que se supunha, e que os desafios a serem enfrentados são mais abrangentes do que os meros aspectos técnicos, normalmente tidos como os únicos relevantes.

Nesta seção apresentam-se os principais conceitos relacionados à reutilização de software, as principais técnicas existentes, e uma discussão mais detalhada sobre a relação entre estes pontos e o processo de software.

2.1.1 Conceitos da reutilização de software

De acordo com Krueger (1992), reutilização de software é o processo de criação de software a partir de software já existente, ao invés de construir do início. O termo *reutilização de software* é comumente confundido com a *reutilização de código*, talvez por ser esta a forma mais simples e melhor compreendida de reutilização (POULIN; CARUSO; HANCOCK, 1993). Porém, a reutilização de código não representa o maior benefício em potencial, pois descarta conhecimento importante que se encontra em outros artefatos, como aqueles produzidos durante análise e projeto.

De fato, qualquer artefato pode ser reutilizado. Segundo D'Souza e Wills (1999), um artefato reutilizável é uma parte do trabalho que pode ser utilizado em mais de um projeto. Nesse sentido, podem ser reutilizados, além do código-fonte, artefatos como código compilado,

casos de teste, modelos, interfaces para usuário, e até mesmo planos e estratégias, entre outros.

Algumas motivações para se reutilizar software são a redução de tempo e esforço no desenvolvimento. Pode-se também aumentar a qualidade do software, reutilizando-se artefatos com qualidade assegurada, o que também acaba por reduzir tempo e esforços na manutenção (KRUEGER, 1992; LIM, 1994). Mas a principal motivação é que reutilização é mais do que apenas uma vantagem que pode ser completamente descartada. Muitas organizações de desenvolvimento de software, mesmo aquelas que alegam não ter preocupação explícita com reutilização, são extremamente dependentes da eficiência com que geram, assimilam e reutilizam seu conhecimento (DESOUZA; AWAZU; TIWANA, 2006).

Estas afirmações levam à seguinte discussão, normalmente associada à reutilização de software: sendo relativamente antiga, com pelo menos quatro décadas, por que a reutilização ainda não é uma prática consagrada, disseminada e bem-sucedida dentro da Engenharia de Software? Indo além, sendo uma prática considerada vital para o sucesso (DESOUZA; AWAZU; TIWANA, 2006), como as organizações vêm sobrevivendo sem empregá-la?

Na verdade, essa discussão ignora um aspecto importante: a reutilização existe desde o surgimento do software em forma armazenada, em 1947, quando Wheeler e Wilkes desenvolveram o conceito de "jump", um precursor do comando "goto", para o computador EDSAC na universidade de Cambridge (OMG, 2003). Desta forma, podiam reutilizar subrotinas pre-construídas para a solução de problemas numéricos comuns. Desde então, programadores efetivamente reutilizam software, seja procurando em seus registros pessoais, projetos antigos, repositórios públicos ou mesmo sua própria memória (DESOUZA; AWAZU; TIWANA, 2006).

Além disso, projetos de software raramente envolvem somente elementos completamente novos. A literatura mostra que entre 40% e 60% do código de uma aplicação pode ser reutilizado em outra aplicação, 60% dos artefatos de projeto são reutilizáveis, 75% das funções são comuns a mais de um programa, e apenas 15% do código de um sistema é único e novo (EZRAN; MORISIO; TULLY, 2002). Outros autores citam taxas de potencial de reutilização que variam entre 15% e 85% (MILI; MILI; MILI, 1995). Ou seja, reutilização é intrínseca ao desenvolvimento de software.

A grande questão, que vem motivando décadas de pesquisa, é que a reutilização não-sistemática, que é realizada de forma *ad hoc*, dependente de iniciativa, conhecimento e talento individuais, não é implantada de forma consistente na organização, e é sujeita a pouco ou nenhum tipo de controle e planejamento gerenciais. Muitas vezes tem efeitos caóticos, alimentando a cultura do "heroísmo" individual e "apagamento de incêndios" e amplificando problemas e defeitos ao invés de reduzi-los (EZRAN; MORISIO; TULLY, 2002).

Dessa forma, o que é necessário é a reutilização sistemática (FRAKES; ISODA, 1994). A reutilização sistemática consiste no entendimento sobre como é possível contribuir para os objetivos de negócio, na definição de estratégias técnicas e gerenciais para se extrair o máximo da reutilização, na integração com processos de software e de melhoria, entre outros aspectos (EZRAN; MORISIO; TULLY, 2002) que fazem com que a reutilização ocorra de forma controlada e repetível.

Porém, colocar essas idéias em prática é uma tarefa complexa, pois apenas agrupar um conjunto de artefatos reutilizáveis em um repositório, disponibilizando-os para os desenvolvedores, não é suficiente. Existe uma série de fatores envolvidos: gerenciais, organizacionais, econômicos, conceituais, legais e técnicos (SAMETINGER, 1997; FRAKES; ISODA, 1994; MORISIO; EZRAN; TULLY, 2002; LUCRÉDIO et al., 2008). Estes fatores tornam muitos casos de sucesso em reutilização, como por exemplo os descritos por Bauer (1993), Endres (1993), Griss (1994, 1995), Joos (1994) mais a exceção do que a regra, e fazem com que a pesquisa na área ainda tenha muitos pontos em aberto.

Esta tese trata da perspectiva técnica da reutilização, mais especificamente os pontos relacionados ao processo de software. Nesse contexto, existem várias abordagens que visam promover a reutilização de software. Apesar de serem distintas, todas elas compartilham quatro conceitos básicos (BIGGERSTAFF; RICHTER, 1989), conforme citados por Krueger (1992):

Abstração: é o princípio essencial da reutilização. Para que um determinado artefato de software possa ser reutilizado, ele precisa antes ser compreendido, de forma que o reutilizador consiga saber se ele irá atender às suas necessidades, se é necessário fazer alguma adaptação, quanto esforço será necessário para essa adaptação, ou se não é possível reutilizá-lo. Sem abstrair os detalhes, o reutilizador gastaria muito tempo examinando cada artefato em busca dessas respostas. Pode-se também pensar na abstração como sendo uma separação entre uma **parte visível** ou abstrata, que contém as informações mais conceituais necessárias à reutilização, e uma **parte escondida**, que contém as informações mais detalhadas, normalmente ligadas à implementação;

Seleção: bibliotecas de software, principalmente em grandes organizações, tendem a ser imensas, envolvendo um grande número e diversos tipos de artefatos que podem ser reutilizados. Encontrar algo de útil em tal cenário é uma tarefa difícil, e uma busca manual pode levar mais tempo do que construir o artefato novamente. Por isso, a maioria das abordagens para reutilização emprega algum tipo de técnica para agilizar a seleção, tais como índices, catálogos, busca automática, entre outras (LUCRÉDIO; ALMEIDA; PRADO, 2004);

Adaptação: a princípio, qualquer artefato pode ser reutilizado em um contexto diferente daquele em que foi projetado inicialmente, uma vez que se saiba que ele irá atender às necessidades do reutilizador. O fator determinante é a quantidade de esforço necessário para adaptar esse artefato para seu novo contexto. Para diminuir esse esforço, o principal foco da maioria das abordagens voltadas à reutilização é tentar criar artefatos genéricos o suficiente para atenderem às necessidades de diversas aplicações possíveis. Esses artefatos são então adaptados para o novo contexto, por meio de parâmetros, arquivos de configuração, ou mesmo pequenas modificações; e

Integração: uma das dificuldades inerentes à reutilização diz respeito à arquitetura do software final, uma vez que ele irá conter pedaços de outros sistemas de software. Quando é necessário integrar artefatos de software que não foram originalmente projetados para operar de forma conjunta, surge uma série de desafios e dificuldades, como por exemplo tentar manter a consistência e padronização das interfaces, prever possíveis necessidades de adaptação ou modificação, e realizar testes de forma a validar a funcionalidade em nível de sistema. O desastre com o foguete europeu Ariane 5 (JEZEQUEL; MEYER, 1997) e os enormes prejuízos decorrentes alertam para a importância deste aspecto.

Estes quatro conceitos básicos estão presentes nas diferentes formas de reutilização, desde o simples processo de se copiar um trecho de código de um local para outro, até as abordagens comumente descritas na literatura, como o desenvolvimento baseado em componentes (SAMETINGER, 1997; SZYPERSKI, 1999), linguagens específicas de domínio (DEURSEN; KLINT; VISSER, 2000), programação generativa (CZARNECKI; EISENECKER, 2000), *frameworks* (JOHNSON, 1997b), padrões (BUSCHMANN et al., 1996) e reengenharia de software (JACOBSON; LINDSTROM, 1991)[1].

2.1.2 O processo de reutilização de software

A simples adoção de uma ferramenta ou técnica não é suficiente para que os benefícios da reutilização sejam alcançados em sua máxima extensão, sendo necessários outros fatores, como um bom gerenciamento organizacional e infraestrutura voltados à reutilização (RINE; SONNEMANN, 1998). Além desses, conforme já discutido no Capítulo 1, o foco no processo é de extrema importância para uma organização em busca da reutilização de software.

Ainda não existe um consenso sobre quais atividades são necessárias para um processo sistemático de reutilização. Existem diversas abordagens distintas, cada uma com um foco

[1] O apêndice A apresenta uma análise mais aprofundada destas abordagens.

maior em determinada parte do processo, procurando solucionar parte do problema. Em termos gerais, um processo de reutilização de software deve definir ao menos duas atividades essenciais: o desenvolvimento **para** reutilização, que consiste no desenvolvimento dos artefatos de software que serão posteriormente reutilizados, e o desenvolvimento **com** reutilização, que consiste no desenvolvimento de aplicações que reutilizam os artefatos previamente desenvolvidos.

Esta distinção entre desenvolvimento para/com reutilização é o ponto fundamental da abordagem de engenharia de software conhecida como *Linha de Produtos de Software* (CLEMENTS; NORTHROP, 2002; LINDEN; SCHMID; ROMMES, 2007), descrita a seguir.

2.1.2.1 Linhas de produtos de software

A origem desta abordagem pode ser rastreada até um trabalho da década de 1970, de Parnas (1976), o mesmo autor que definiu os conceitos de encapsulamento da informação, um dos fundamentos da orientação a objetos. Neste trabalho é descrito o conceito de famílias de programas, e apesar de inicialmente ter focado na variabilidade em termos das características não funcionais, Parnas estabelece a base para as linhas de produto de software (LINDEN; SCHMID; ROMMES, 2007).

O princípio definido era o de estudar primeiro o que os programas de uma mesma família tinham em comum, para depois tentar entender o que os diferenciava. Em seguida, duas alternativas de método produziam programas incompletos que implementavam as partes comum da família, e deixavam as partes variáveis para serem instanciadas posteriormente: o método de refinamento por etapas (*stepwise refinement*), que produzia um programa no qual alguns tipos de operandos e operadores eram deixados sem implementação; e o método de especificação dos módulos, que baseava-se na especificação em alto nível de unidades de comportamento de programas e no encapsulamento da informação para que o restante do código pudesse ser idealizado e construído. Para construir um novo programa, um desenvolvedor reutilizava este programa incompleto e o completava de acordo com as variabilidades desejadas, seja providenciando os operadores e operandos, ou efetivamente implementando os módulos especificados.

Ou seja, há uma mudança de foco. Ao invés de desenvolver software considerando os requisitos de um único sistema, tenta-se desenvolver software que consiga atender aos requisitos de um conjunto de sistemas similares, de forma que é possível reaproveitar as partes comuns e apenas desenvolver o que é completamente novo. Na nomenclatura típica da área de linhas de produtos de software, este conjunto de sistemas similares é chamado de família de produtos, ou

domínio de aplicações. As partes reutilizáveis são a arquitetura de referência e os artefatos do núcleo (*core assets*). O desenvolvimento das partes comuns (desenvolvimento para reutilização) é chamado de *Engenharia de Domínio* e o desenvolvimento de um produto (desenvolvimento com reutilização), de *Engenharia de Aplicações* (CLEMENTS; NORTHROP, 2002; MILI et al., 2002; LINDEN; SCHMID; ROMMES, 2007).

Com exceção de uma preocupação maior com aspectos de negócio e alguns avanços em termos de métodos, os conceitos apresentados por Parnas são praticamente os mesmos que fundamentam as linhas de produtos (LINDEN; SCHMID; ROMMES, 2007):

1. **Gerenciamento da variabilidade**: consiste na determinação, modelagem e implementação das comunalidades e variabilidades de uma família de produtos (programas). Equivale à análise das semelhanças e diferenças entre programas de uma mesma família;

2. **Definição arquitetural**: uma arquitetura de referência tem papel chave na linha de produtos, oferecendo meios concretos para que diferentes produtos possam ser instanciados. Equivale à definição de quais pontos de um programa devem ser deixados sem implementação, de acordo com as variabilidades identificadas; e

3. **Abordagem em dois ciclos**: consiste na divisão entre engenharia de domínio e engenharia de aplicações, de forma equivalente ao refinamento por etapas ou especificação dos módulos e à criação dos programas.

O grande diferencial das abordagens atuais é a preocupação em definir o escopo da família de produtos de acordo com objetivos de negócio e estratégias de mercado, visando vantagens a longo prazo. Em outras palavras, define-se quais produtos devem ser desenvolvidos, quais as características mais importantes a serem implementadas, entre outras questões (LINDEN; SCHMID; ROMMES, 2007). Esta preocupação faz com que as linhas de produtos sejam atrativas para a indústria, que já acumula diversos relatos de sucesso, conforme pode ser constatado no *Hall* da Fama em linhas de produto[2], que conta com empresas como Philips, Boeing, Lucent, entre outras que conseguiram adotar esta prática com sucesso.

2.1.2.2 Elementos de um processo de reutilização

Apesar da falta de consenso, um estudo preliminar (GARCIA et al., 2007, 2008) analisou diversos trabalhos e modelos descritos na literatura, buscando identificar quais elementos de

[2] `http://www.sei.cmu.edu/productlines/plp_hof.html`

processo são necessários para que a reutilização possa ser bem sucedida. Este estudo incluiu os esforços do SEI/CMU (*Software Engineering Institute / Carnegie Mellon University*) (DOE; BERSOFF, 1986; PYSTER; BARNES, 1988; HOLIBAUGH et al., 1989), mais focados na área de custos relacionados à reutilização; do SPC (*Software Productivity Consortium*, primeiro com o RMM (*Reuse Maturity Model*) (KOLTUN; HUDSON, 1991), depois com o RPC (*Reuse Capability Model*) (DAVIS, 1993) e finalmente com o RRM (*Reuse Reference Model*) (RINE; NADA, 2000), modelos que descrevem as principais práticas de reutilização em uma organização de software; e de outros importantes autores da área, como Prieto-Díaz (1991, 1993), Sherif, Appan e Lin (2006) e Morillo et al. (2006). No estudo de Garcia et al. (2007, 2008) foi definido um modelo que reúne as práticas comumente relacionadas à reutilização de software.

O modelo inicial deste estudo (Figura 1) é baseado em 18 áreas de processo (AP) que descrevem as atividades essenciais para uma organização que deseja incorporar a reutilização sistemática em seu processo.

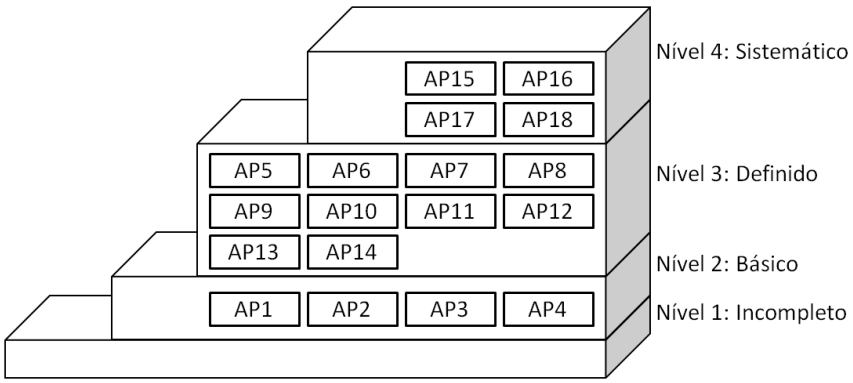


Figura 1: Modelo de maturidade em reutilização (GARCIA et al., 2007, 2008)

As áreas de processo estão agrupadas em quatro níveis, descritos a seguir:

- **Nível 1 - Incompleto**: neste nível, apenas o desenvolvimento de software tradicional é realizado. Práticas de reutilização são usadas esporadicamente ou mesmo ignoradas e desencorajadas pela gerência. Reutilização é fruto de esforço individual, e os custos da reutilização são desconhecidos;

- **Nível 2 - Básico**: este nível é caracterizado pela utilização básica de artefatos com potencial de reutilização. Engloba algumas atividades iniciais orientadas à reutilização, como a *manutenção de métodos e técnicas básicas de reutilização (AP1)*, o *planejamento da reutilização (AP2)*, a *definição dos objetivos da reutilização (AP3)* e a *implementação do domínio (AP4)*, porém ainda sem a preocupação com análise e projeto voltados à reutilização;

- **Nível 3 - Definido**: neste nível o principal foco de engenharia é o suporte à variabilidade. Enquanto no nível 2 a preocupação era aumentar o nível de reutilização de artefatos individuais, aqui o foco é oferecer um suporte global à variabilidade do domínio, principalmente com as práticas de *controle e monitoramento do processo de reutilização (AP5)*, *integração com o ciclo de vida do software (AP6)*, *análise e projeto do domínio (AP7 e AP8)*. Também neste nível introduz-se a preocupação com a área de qualidade, incluindo o *treinamento em reutilização (AP9)*, *gerenciamento de uma unidade de reutilização (AP10)*, *programas de métricas e de avaliação organizacional (AP11 e AP12)* e *avaliação da qualidade dos artefatos (AP13)*. É também neste nível que se encontram as práticas ligadas à *engenharia de aplicações (AP14)*, foco do desenvolvimento **com reutilização**.

- **Nível 4 - Sistemático**: este é o nível mais avançado que uma organização pode chegar em termos de reutilização de software. Neste nível, o processo está em constante *otimização (AP15)*, de acordo com resultados de projetos anteriores. Também aqui a qualidade dos artefatos reutilizáveis é sujeita a um processo mais rigoroso de *controle de qualidade (AP16)*. Outras práticas interessantes deste nível 4 incluem a tentativa de se *prever e suprir necessidades futuras em termos de artefatos reutilizáveis (AP17)*, e uma *análise de mercado para se avaliar as questões de investimento em reutilização (AP18)*;

Este modelo busca identificar as práticas essenciais sem entrar em detalhes sobre como as mesmas são realizadas. O modelo também não define quais práticas são obrigatórias, quais são opcionais, quais são apenas desejáveis, etc. Fica a cargo da organização decidir quais práticas adotar, de acordo com seu contexto, e a melhor maneira de realizar cada prática.

Nesta tese foram realizadas somente as práticas relacionadas ao desenvolvimento **para** reutilização, aqui descritos como análise, projeto e implementação do domínio. O objetivo foi dar suporte às atividades essenciais, relacionando cada uma com o desenvolvimento orientado a modelos e como o mesmo pode ajudar na obtenção dos objetivos de cada prática. Mais detalhes sobre como este modelo se relaciona com a presente abordagem são apresentados no Capítulo 4.

2.2 Desenvolvimento orientado a modelos

Apesar dos inúmeros avanços na área da Engenharia de Software, ainda existem problemas (KLEPPE; WARMER; BAST, 2003) com a maneira com que o software é desenvolvido na maioria das organizações, que permanecem como desafios até os dias atuais (FRANCE; RUMPE, 2007). Conforme discutido brevemente no Capítulo 1, esses problemas derivam do fato de o software

ser atualmente um artefato extremamente complexo. Dentre esses problemas, destacam-se os sete descritos a seguir.

O fardo da modelagem: arquitetos de software (algumas vezes) usam UML (*Unified Modeling Language* ou Linguagem de Modelagem Unificada) (OMG, 2007) para criar modelos de alto nível, que são úteis em um primeiro momento, para facilitar a análise de um problema. Através de modelos, facilita-se a realização de discussões, trocas de idéias e comunicação entre as equipes envolvidas com o processo de software.

Porém, esses modelos logo se tornam inúteis, à medida que o desenvolvimento avança. Isto porque programadores, que criam código manualmente, também realizam mudanças e manutenções diretamente no código. Desta forma, sem um trabalho extra para atualizá-los, os modelos logo perdem a consistência, se tornando incapazes de representar a realidade.

Mesmo com técnicas de engenharia reversa, que facilitam a extração automática de modelos a partir do código, a verdade é que os modelos são artefatos desnecessários, no sentido em que não fazem parte do software propriamente dito. São apenas parte da documentação, que precisa ser atualizada com esforço não diretamente produtivo, tornando-se literalmente um fardo a ser carregado pela equipe.

Além disso, modelos são mais úteis para membros novos de uma equipe. Nos projetos em que uma mesma equipe segue trabalhando por um longo tempo, as informações nos modelos já são bem conhecidas pelos membros da equipe, e portanto nem valor de documentação os mesmos possuem. É bastante comum, na chegada de um novo membro a uma equipe, o mesmo perguntar pela documentação e modelagem do sistema. E é também comum este membro obter como resposta o fato de que os documentos estão antigos e desatualizados.

Reutilização de conhecimento: além de ser um conjunto de linhas ordenadas de instruções e comandos que expressam um algoritmo para um computador, o código-fonte encapsula conhecimento de uma forma concreta. Algoritmos, estruturas de dados, classes e funções representam meses ou anos de evolução de um software, e ali estão contidas as experiências da equipe, uma tradução dos requisitos do software em uma forma executável e altamente codificada.

Este conhecimento muitas vezes representa apenas instruções de baixo nível, que servem para operação do computador, manipulação de variáveis, truques de otimização, entre outros. Mas ele também é um retrato das regras de negócio do software, sendo importante, e às vezes vital para o sucesso da organização.

O problema é que a linguagem do código-fonte é demasiadamente densa e codificada, de

forma que é difícil identificar, extrair e reutilizar este conhecimento apenas lendo este código. Além disso, este conhecimento está impregnado por trechos altamente associados a detalhes de plataforma, de modo que a reutilização de uma determinada lógica de negócio em uma plataforma ou linguagem diferentes requer um trabalho cuidadoso de reengenharia.

São necessárias formas mais intuitivas de representação do conhecimento, que sejam menos dependentes do código-fonte. Uma alternativa é o uso de documentação apropriada, mas como já foi discutido na seção anterior, existe o problema da inconsistência causada por modificações no software.

Produtividade: o desenvolvimento de software é normalmente constituído de projeto de baixo nível e codificação. Artefatos de alto nível (modelos, diagramas) são produzidos antes da codificação, e servem de auxílio às tarefas de desenvolvimento e manutenção, mas não constituem o software propriamente dito.

Além disso, esses artefatos logo perdem seu valor e se tornam retratos irreais do software assim que as fases de codificação começam, pois quaisquer eventuais mudanças que o software sofra são realizadas somente no código e não são refletidas nos artefatos de alto nível, devido principalmente às restrições de tempo. Sendo assim, o tempo gasto na construção desses artefatos, assim como o tempo gasto em esforços de manutenção devido a essa inconsistência com o código, não são diretamente aproveitados na produção do software.

Outro ponto referente à produtividade diz respeito à multiplicidade entre elementos mais conceituais e as linhas de código. A um único elemento conceitual podem corresponder inúmeras linhas de código. Por exemplo, considere uma máquina de estados. Um único estado pode estar representado no código em diferentes locais, incluindo tabelas de transições, variáveis locais e métodos. Suponha que cada estado possui 50 linhas de código associadas. Dessa forma, para se inserir ou modificar um estado, 50 linhas de código precisam ser inseridas ou modificadas.

Além disso, essa multiplicidade não é uma função objetiva, ou seja, não se pode garantir que a todos os estados correspondam sempre 50 linhas. A variação disso torna o mapeamento entre elementos de alto nível nos seus respectivos códigos um trabalho mais exaustivo.

Como resultado, muitas tarefas de implementação são repetitivas e exaustivas, gastando esforço que poderia ser melhor aproveitado em trabalho conceitual.

Manutenção e documentação: criar documentação correta e atualizada é uma das tarefas mais essenciais para que um sistema de software sobreviva e evolua de maneira efetiva. Porém, criar e manter a documentação atualizada é normalmente uma tarefa manual não muito

apreciada por desenvolvedores.

É possível obter a documentação diretamente a partir do código, usando técnicas automatizadas de engenharia reversa. Porém, essa documentação seria apenas um reflexo do código, e não uma documentação de alto nível, que muitas vezes é crucial para as tarefas de manutenção em sistemas muito complexos. Nesses casos, o trabalho manual se mostra como a única solução.

Além disso, os mesmos problemas relacionados à produtividade citados na seção anterior possuem impacto na manutenção. Por exemplo, modificar um único campo de uma aplicação baseada em Struts[3] pode requerer: alteração das tabelas, índices, visões, consultas e procedimentos armazenados que possuem relação com o campo; alteração das classes Java correspondentes às ações de consulta, inserção e atualização que envolvem o campo em questão; alteração do mapeamento de entidades (*beans*) que contém referências para este campo, normalmente um arquivo XML; alteração dos arquivos de descrição dos formulários que contém referências para este campo, normalmente arquivos XML; entre outras. Ou seja, o mesmo trabalho braçal e repetitivo que foi necessário para a construção do software é necessário também na manutenção.

Outro problema causado na manutenção diz respeito à rastreabilidade entre elementos conceituais e elementos de implementação. Ao se planejar uma mudança, primeiro pensa-se em termos conceituais, para apenas em seguida identificar os trechos de código a serem modificados. Sem esta informação de rastreabilidade, esta tarefa é dificultada, exigindo uma busca cuidadosa no código.

Também pode levar a erros de inconsistência entre os diversos artefatos relacionados. Por exemplo, a modificação de código Java sem a modificação correspondente dos comandos SQL pode causar erros de execução.

Validação e otimização: encontrar erros conceituais diretamente no código-fonte é uma tarefa mais difícil e trabalhosa do que se fossem utilizados modelos mais conceituais. Por exemplo, considere uma máquina de estados com centenas de estados. Identificar, olhando apenas para o código, a existência de estados inalcançáveis, transições infinitas ou estados "mortos" (estados sem transições para fora) pode ser extremamente trabalhoso.

Da mesma forma, otimizações mais conceituais podem não ser tão simples de serem executadas olhando-se diretamente para o código. Por exemplo, a remoção de estados redundantes em uma máquina de estados seria facilitada caso houvesse algum tipo de artefato de mais alto nível associado.

[3] `http://jakarta.apache.org/struts/`

Enquanto documentos de alto nível poderiam ser utilizados nestas tarefas, os mesmos problemas descritos anteriormente como fardos da modelagem teriam de ser enfrentados. Além disso, a validação e otimização automáticas, que em muitos casos são superiores a um processo manual, requerem modelos formais consistentes com a implementação, caso contrário os resultados seriam inexatos ou mesmo equivocados.

Portabilidade: a indústria de software é extremamente dinâmica, e novas tecnologias e plataformas surgem muito rapidamente, oferecendo vantagens que forçam as organizações a se adaptarem rapidamente para não ficarem desatualizadas em relação aos principais concorrentes. O processo de reengenharia necessário para portar o software para essas plataformas é dispendioso e muitas vezes demorado.

Por outro lado, o surgimento de novas tecnologias pode aumentar a pressão existente para a migração, e a estagnação pode ser prejudicial para a organização, que se vê obrigada a realizar a migração ou permanecer defasada em relação aos concorrentes.

O problema da portabilidade surge da quantidade de esforço despendido em tarefas específicas da plataforma, esforço este que não pode ser reaproveitado em outras plataformas. Idealmente, o desenvolvimento de software deveria ser mais conceitual e menos focado em esforço repetitivo.

Interoperabilidade: sistemas de software raramente funcionam isoladamente. Normalmente eles precisam se comunicar entre si, para trocar informações ou realizar tarefas em conjunto. Além disso, com as atuais idéias de componentes de software, um mesmo sistema pode ser composto de partes que utilizam tecnologias diferentes, e que ainda assim precisam se comunicar, normalmente em ambientes heterogêneos, como a Internet.

Neste contexto, a interoperabilidade torna-se um requisito do software, trazendo consigo vários outros, como a necessidade de equipes distintas, com profissionais e especialistas dedicados às diferentes partes do software. Também exige formas de comunicação mais eficientes entre as diferentes equipes e a gerência, já que cada pessoa envolvida nestes projetos multidisciplinares tem conhecimentos e interesses em diferentes partes do problema.

Uma possível solução: o MDD surgiu como uma maneira de se resolver estes problemas, utilizando uma abordagem altamente automatizada e com promessas de ganhos significativos em termos de qualidade e produtividadeA seguir apresenta-se os principais conceitos e técnicas envolvidas com o MDD.

2.2.1 Conceitos do desenvolvimento orientado a modelos

O desenvolvimento de software orientado a modelos (MDD - *Model-Driven Development* surgiu com o objetivo de ajudar a resolver os problemas citados na seção anterior (KLEPPE; WARMER; BAST, 2003). O MDD é também conhecido como MDE (*Model-Driven Engineering*) (SCHMIDT, 2006), MDSD (*Model-Driven Software Development*) (VÖLTER; GROHER, 2007) ou, para aqueles cansados de tantos acrônimos, MD* (VÖLTER, 2008). Todos esses acrônimos dizem respeito à mesma abordagem, cuja idéia principal é reconhecer a importância dos modelos no processo de software, não apenas como um "guia" para tarefas de desenvolvimento e manutenção, mas como parte integrante do software.

A proposta do MDD (Figura 2) é fazer com que o engenheiro de software não precise interagir manualmente com todo o código-fonte, podendo se concentrar em modelos de mais alto nível, ficando protegido das complexidades requeridas para implementação nas diferentes plataformas. Um mecanismo automático é responsável por gerar automaticamente o código a partir dos modelos. Neste cenário, modelos não são apenas um guia, ou uma referência. Eles fazem parte do software, assim como o código-fonte.

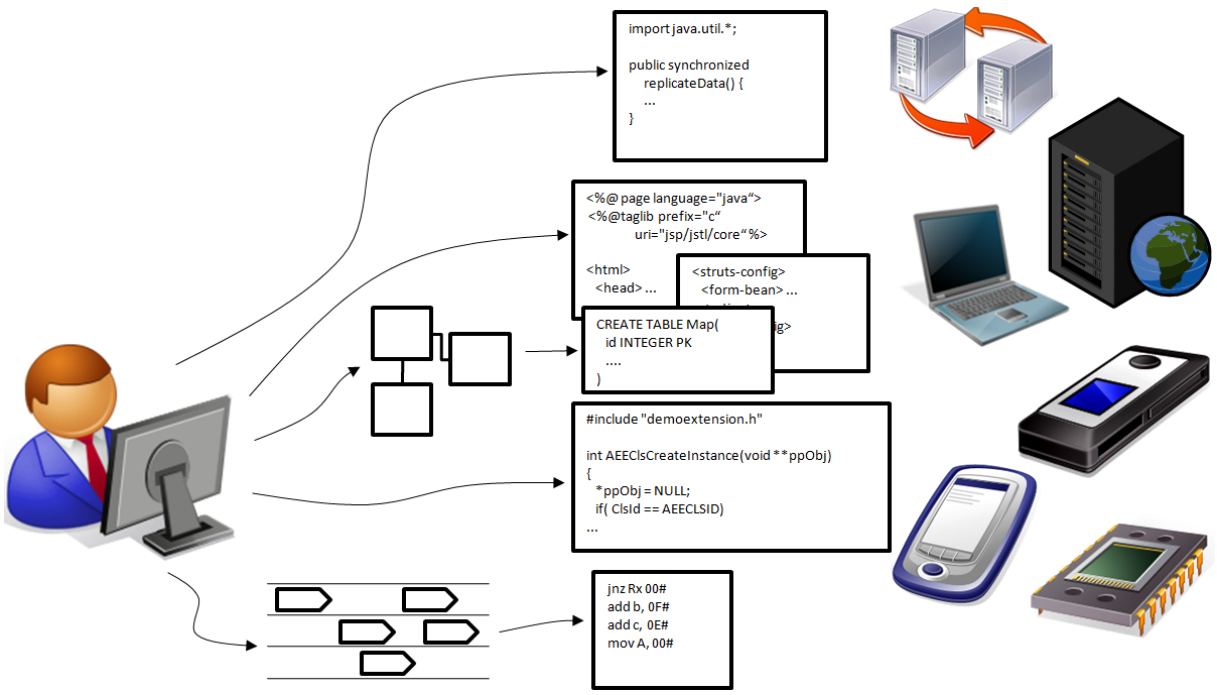


Figura 2: Ilustração do processo de criação de software no desenvolvimento orientado a modelos

2.2.1.1 Principais vantagens e desvantagens

As principais vantagens da abordagem MDD relacionam-se com os sete problemas destacados anteriormente. As vantagens, apresentadas por Kleppe, Warmer e Bast (2003), Deursen e Klint (1998), Bahnot et al. (2005), Mernik, Heering e Sloane (2005), são:

- **Produtividade**: o tempo de desenvolvimento será melhor aproveitado, pois será gasto na produção de modelos de mais alto nível. Tarefas repetitivas podem ser implementadas nas transformações, poupando tempo e esforço que podem ser despendidos em tarefas mais importantes. Além disso, um único modelo pode gerar uma grande quantidade e diversidade de código;

- **Portabilidade**: um mesmo modelo pode ser transformado em código para diferentes plataformas;

- **Interoperabilidade**: cada parte do modelo pode ser transformado em código para uma plataforma diferente, resultando em um software que executa em um ambiente heterogêneo, porém mantendo a funcionalidade global. Também podem ser gerados adaptadores e conectores utilizando tecnologias independentes de plataforma, para que esses códigos de diferentes plataformas possam se comunicar;

- **Manutenção e documentação**: no desenvolvimento convencional, a urgência inerente às atividades de manutenção faz com que os desenvolvedores insiram modificações diretamente no código, fazendo com que a documentação fique logo desatualizada. No MDD os modelos não são abandonados. Eles fazem parte do software, e as alterações são realizadas diretamente neles, mantendo-os consistentes com o código. Desta forma, a documentação mantém-se atualizada, o que acaba por facilitar as tarefas de manutenção;

- **Comunicação**: no MDD, os diferentes profissionais possuem meios mais efetivos para comunicação, uma vez que modelos geralmente são mais abstratos que o código, não exigindo conhecimento técnico relativo à plataforma. Especialistas do domínio têm um papel mais ativo no processo, podendo utilizar diretamente os modelos para identificar mais facilmente os conceitos do negócio, enquanto especialistas em computação podem identificar os elementos técnicos;

- **Reutilização**: Diversos autores, tais como Krueger (1992), Griss (1995), Frakes e Isoda (1994), Jacobson, Griss e Jonsson (1997), ressaltam que a reutilização de artefatos de alto nível proporciona maiores benefícios do que a reutilização de código-fonte. No desenvolvimento convencional, reutilizar um modelo requer a transformação manual para

reutilizar também o código a ele associado. No MDD, isto é mais facilmente alcançado, pois o código pode ser automaticamente re-gerado para o novo contexto;

- **Verificações e otimizações**: os modelos oferecem mais munição para verificadores semânticos e otimizações automáticas específicas de domínio poderem ser executados. Isto pode reduzir a ocorrência de erros semânticos e prover implementações mais eficientes;

- **Corretude**: além do fato de geradores não introduzirem erros acidentais, como erros de digitação, geradores permitem que a identificação de erros conceituais aconteça em um nível mais alto de abstração.

Em resumo, as vantagens do MDD derivam da capacidade de evitar que o desenvolvedor precise executar as tarefas repetitivas necessárias para a transformação de modelos para o código final executável. Isto é alcançado por meio da automação dessas transformações. O tempo gasto nessas tarefas, que no desenvolvimento convencional (não orientado a modelos) inibe a execução do ciclo completo dos requisitos aos testes, é significativamente reduzido, fazendo com que mesmo atividades de urgência, como correção de erros, possam ser executadas sem produzir inconsistência com os modelos, mantendo-os sempre atualizados.

Entre as desvantagens causadas pelo MDD, alguns autores, como Ambler (2003), Thomas (2004), citam as seguintes:

- **Rigidez**: o MDD causa maior rigidez no software produzido, uma vez que grande parte do código é gerado e fica além do alcance do desenvolvedor;

- **Complexidade**: os artefatos necessários para uma abordagem baseada em modelos, como por exemplo ferramentas de modelagem, transformações e geradores de código, introduzem complexidade adicional ao processo, pois tratam-se de artefatos inerentemente mais difíceis de construir e manter;

- **Desempenho**: apesar de algumas otimizações poderem ser realizadas em nível mais alto de abstração, a regra geral é que geradores acabam incluindo muito código desnecessário e, portanto, o resultado pode não apresentar desempenho ótimo, quando comparado com código escrito à mão;

- **Curva de aprendizado**: o desenvolvimento dos artefatos específicos do MDD exigem profissionais com habilidades na construção de linguagens, ferramentas de modelagem, transformações e geradores de código. O aprendizado destas técnicas, apesar de não ser extremamente difícil, requer um treinamento dedicado; e

- **Alto investimento inicial**: assim como a reutilização de software, o MDD depende de maior investimento inicial, uma vez que a construção de uma infraestrutura de reutilização orientada a modelos requer mais tempo e esforço. Porém, os ganhos posteriores são significativos, fazendo com que este investimento tenha retorno em poucas interações.

2.2.1.2 Principais elementos do MDD

Obviamente, automatizar as transformações não é uma tarefa simples. A Figura 3 mostra os principais elementos necessários para essa abordagem, e como eles são combinados.

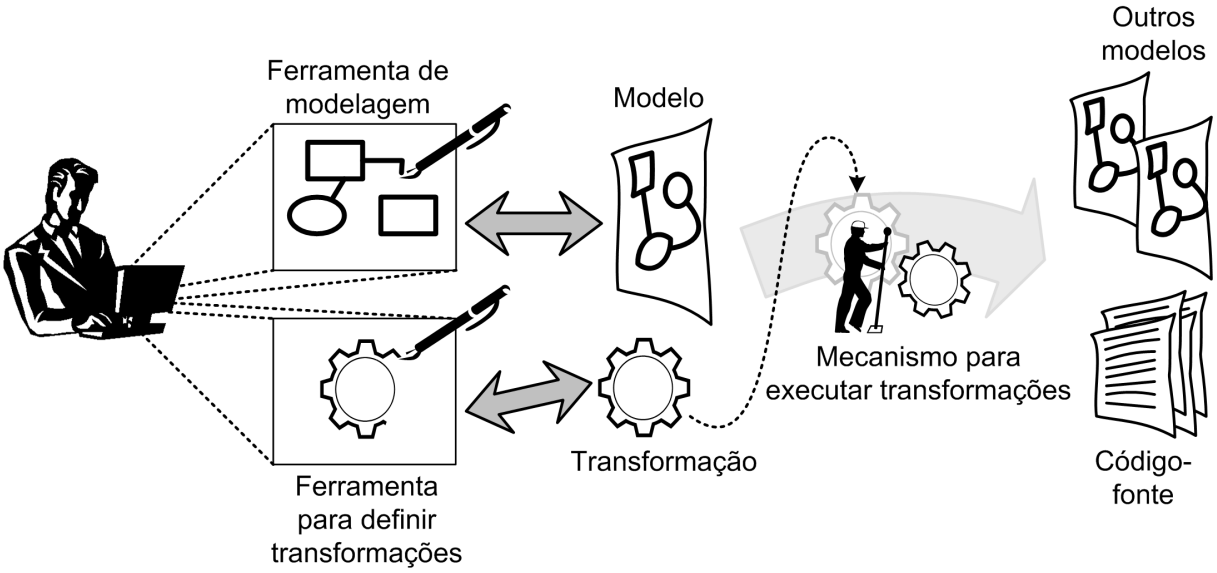


Figura 3: Principais elementos do MDD

Para possibilitar a criação de modelos, é necessária uma ferramenta de modelagem. Utilizando essa ferramenta, o engenheiro de software produz modelos que descrevem os conceitos do domínio. Para isto, a ferramenta deve ser intuitiva e de fácil utilização. Ao mesmo tempo, os modelos por ela criados precisam ser semanticamente completos e corretos, uma vez que devem poder ser compreendidos por um computador, e não um ser humano, capaz de corrigir pequenos enganos ou preencher lacunas por si só. O elemento que implementa estas características é normalmente uma linguagem específica de domínio, ou DSL (*Domain-Specific Language*), descrito no próximo capítulo.

Os modelos servem de entrada para transformações que irão gerar outros modelos ou o código-fonte. Para definir as transformações, é necessária uma ferramenta que permita que o engenheiro de software construa regras de mapeamento de modelo para modelo, ou de modelo para código. Idealmente, essa ferramenta deve possibilitar que as regras de mapeamento sejam construídas da forma mais natural possível, uma vez que a construção de transformadores é uma

tarefa complexa.

Finalmente, é necessário um mecanismo que efetivamente aplique as transformações definidas pelo engenheiro de software. Esse mecanismo deve não só executar as transformações, mas também manter informações de rastreabilidade, possibilitando saber a origem de cada elemento gerado, seja no modelo ou no código-fonte.

Atualmente existem diversos trabalhos que buscam melhor definir o papel de todos esses elementos. A seguir são apresentadas as principais abordagens existentes na indústria para possibilitar o desenvolvimento orientado a modelos.

2.2.2 Principais abordagens da indústria para MDD

Para dar suporte a diferentes linguagens de modelagem, ajudar a garantir que os modelos construídos estejam semanticamente corretos e completos, e possibilitar a definição e execução de transformações genéricas, as principais abordagens existentes na indústria para MDD se baseiam no conceito de metamodelagem (OMG, 2006b), apresentado na Figura 4.

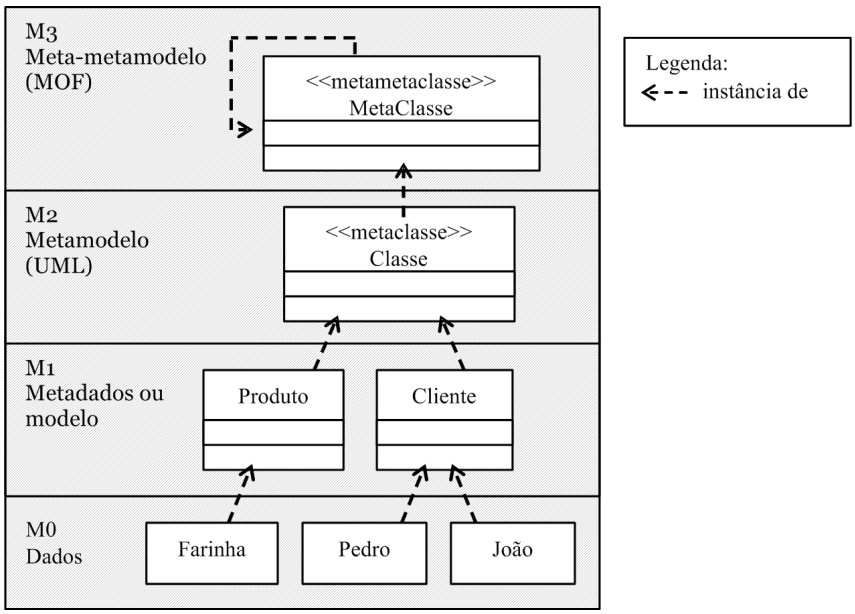


Figura 4: Arquitetura clássica de metamodelagem

O primeiro nível (M0) corresponde aos dados propriamente ditos. O segundo nível (M1) corresponde aos metadados, ou modelo. São os dados que descrevem os dados. O terceiro nível (M2) é o metamodelo, utilizado para a definição de modelos. A especificação UML é um exemplo de metamodelo. O quarto nível (M3) é utilizado para definir metamodelos, ou seja, um meta-metamodelo define linguagens de modelagem, como a UML, por exemplo. Um exemplo de meta-metamodelo é o padrão MOF, apresentado na próxima seção. Não existe um quinto

nível, pois o meta-metamodelo é normalmente instância de si mesmo.

Existem diferentes meta-metamodelos utilizados na indústria, utilizados nas diferentes ferramentas de modelagem e transformação, para uniformizar a manipulação de modelos e código em diferentes linguagens.

Nas seções seguintes são apresentadas as principais abordagens da indústria para o MDD.

2.2.2.1 Abordagem OMG: MDA (*Model-Driven Architecture*)

O OMG (*Object Management Group*) é um dos responsáveis pelo atual aumento de interesse nas idéias do MDD, devido principalmente à MDA. A MDA surgiu como um conjunto de padrões voltados para fabricantes de ferramentas interessados em manter a interoperabilidade com outros fabricantes (OMG, 2003).

A MDA introduz os conceitos de CIM (*Computation Independent Model* ou Modelo Independente de Computação), PIM (*Platform Independent Model* ou Modelo Independente de Plataforma) e PSM (*Platform-Specific Model* ou Modelo Específico de Plataforma).

Um CIM é uma visão do sistema de um ponto de vista que não depende de computação. Um CIM não mostra detalhes da estrutura dos sistemas. É também conhecido como modelo do domínio, e utiliza um vocabulário familiar aos profissionais e especialistas no domínio em questão (OMG, 2003).

Um PIM é uma visão do sistema de forma independente da plataforma de implementação. Essa independência de plataforma, no entanto, chega até um certo nível, de forma que o mesmo modelo possa ser reaproveitado em diferentes plataformas do mesmo tipo (OMG, 2003).

Um PSM é uma visão do sistema do ponto de vista de uma plataforma específica. Um PSM combina as especificações de um PIM com detalhes sobre como o sistema utiliza aquele tipo particular de plataforma (OMG, 2003).

Na MDA, transformações são utilizadas para transformar um modelo em outro, sucessivamente, até o código-fonte. A transformação entre CIM e PIM é menos passível de automação, pois envolve mais decisões e maiores possibilidades de interpretação dos requisitos. Já a transformação entre PSM e código-fonte é mais passível de automação, já que o PSM está intimamente ligado com a plataforma de implementação.

A base da MDA é o MOF (*Meta-Object Facility*) (OMG, 2006b), o meta-metamodelo que serve de base para a definição de todas as linguagens de modelagem. É devido ao MOF que as ferramentas de modelagem e transformação podem trabalhar em conjunto.

O MOF consiste em um padrão orientado a objetos que permite a definição de classes com atributos e relacionamentos, sendo bastante semelhante ao diagrama de classes da UML. Também define uma interface padrão de acesso aos dados dos modelos, oferecendo métodos para manipulação dos dados e dos metadados. Além dessa interface padrão, que é única para qualquer metamodelo, o MOF define regras para a criação de interfaces específicas para cada metamodelo. Por exemplo, para cada atributo monovalorado de uma classe do metamodelo, deve existir um método do tipo *set* e um método do tipo *get*. Essas regras se estendem para nomes de classes, relacionamentos, e assim por diante.

Atualmente, o MOF encontra-se na versão 2.0 (OMG, 2006b). Assim como a UML, é um padrão elaborado e mantido por diversas empresas associadas ao OMG.

A MDA também define o XMI (*XML Metadata Interchange*) (OMG, 2006c), um formato para representar modelos em XML. Este formato define uma estrutura de documento que considera a relação entre os dados e seus correspondentes metadados. Assim, é possível para uma ferramenta, ao interpretar este formato, identificar quais os metadados que descrevem os dados sendo lidos. Diferentes metaníveis podem ser representados, desde o M0 até o M3 (Figura 4). Por exemplo, é possível representar modelos UML (com referência para o metamodelo UML) ou modelos E-R (com referência para o metamodelo E-R). O metamodelo UML também pode ser descrito em XMI, e neste caso teria uma referência para o meta-metamodelo MOF. Por ser baseado em XML, traz consigo várias vantagens, tais como a facilidade de ser interpretado e a possibilidade de se aplicar transformações. Atualmente, o padrão XMI encontra-se na versão 2.1 (OMG, 2006c).

Outro padrão da MDA é o QVT (*Queries/Views/Transformations*) (OMG, 2005). Ainda em fase de finalização, o QVT define uma linguagem textual e uma notação para definir consultas e transformações baseadas no MOF. Por meio dessa linguagem, é possível definir regras de mapeamento entre modelos escritos em qualquer linguagem de modelagem, desde que essa seja uma instância do MOF.

Apesar de ter a proposta de ser genérica, podendo trabalhar com qualquer linguagem de modelagem, a MDA tem grande foco na UML (*Unified Modeling Language*) (OMG, 2007). Por exemplo, perfis UML (OMG, 2006a) são o mecanismo preferido para representação de conceitos específicos de plataforma em um PSM. Mas sendo também instância do MOF, a UML é apenas uma das possíveis linguagens de modelagem que podem ser usadas no MDD.

2.2.2.2 Abordagem Sun Microsystems

Devido ao presente cenário competitivo vivido pelos fabricantes de ferramentas e ambientes de apoio ao desenvolvimento, a Sun Microsystems, principal responsável pela tecnologia Java, tem especial interesse no desenvolvimento baseado em modelos. Como a maioria de seus competidores, a Sun, por meio de sua plataforma NetBeans, busca oferecer ao desenvolvedor a opção de se trabalhar simultaneamente no modelo e código-fonte, seguindo as idéias do MDD. Com essa motivação, a Sun concentra seus esforços em oferecer ferramentas que sigam os padrões OMG, principalmente com duas iniciativas:

- **JMI (*Java Metadata Interface*[4])**: trata-se de um mapeamento das interfaces MOF para a linguagem Java, de forma a possibilitar a manipulação de modelos compatíveis com o MOF utilizando programação Java; e

- **MDR (*MetaData Repository*[5])**: é um subprojeto NetBeans, e consiste em um mecanismo genérico de manipulação de modelos, como o discutido no início deste capítulo. Com base no padrão MOF, é capaz de importar/exportar modelos segundo o padrão XMI, e disponibiliza interfaces no padrão JMI para que os modelos sejam manipulados.

O funcionamento do MDR é ilustrado na Figura 5. Uma ferramenta baseada no MDR pode implementar diferentes funcionalidades, tais como edição visual de modelos, geração de código, entre outras, beneficiando-se das interfaces padrão JMI. Parte dessas funcionalidades é genérica, permanecendo a mesma para qualquer metamodelo, enquanto outra parte é criada de forma específica para cada metamodelo, conforme as regras do MOF. O MDR implementa diretamente a parte genérica, enquanto a parte específica é automaticamente gerada a partir da descrição do metamodelo em um arquivo no formato XMI. Desta forma, o MDR é capaz de criar e manipular modelos em qualquer linguagem de modelagem, desde que seu metamodelo seja instância do MOF.

Internamente, os modelos podem ser mantidos em memória RAM, em um banco de dados, ou no sistema de arquivos do sistema operacional, em uma estrutura de árvore B. O MDR também persiste os modelos no formato XMI, possibilitando o intercâmbio de modelos entre ferramentas e desenvolvedores.

[4] `http://java.sun.com/products/jmi/`
[5] `http://mdr.netbeans.org/`

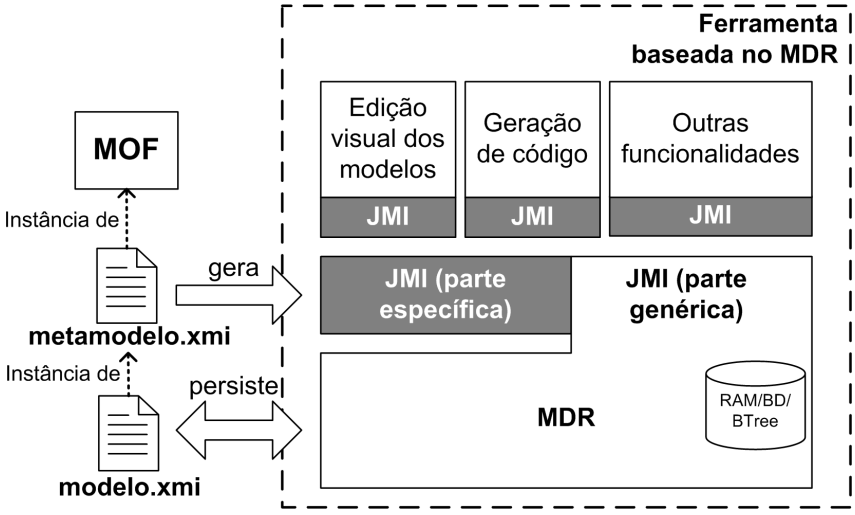


Figura 5: Funcionamento do MDR

2.2.2.3 Abordagem Eclipse

A abordagem liderada pela IBM é baseada na plataforma Eclipse. Uma das bases dessa abordagem é o EMF (*Eclipse Modeling Framework*[6]). O EMF permite a manipulação de modelos segundo seu correspondente metamodelo. O EMF segue um meta-metamodelo denominado *Ecore*, ao invés do MOF, padrão estabelecido pelo OMG. O *Ecore* surgiu como uma implementação do MOF, mas evoluiu para uma forma mais eficiente, a partir da experiência obtida após alguns anos na construção de ferramentas (ECLIPSE, 2005).

Pelo mesmo motivo de eficiência, o EMF não segue fielmente o padrão JMI, mais voltado para repositórios persistentes. O EMF utiliza um subconjunto de mapeamentos de metamodelo para Java otimizados para manipulação em memória (ECLIPSE, 2006), o que o torna, além de mais eficiente, mais simples de ser utilizado, porém menos abrangente do que o conjunto mais completo formado pela combinação JMI/MOF (MOORE et al., 2004).

Além do metamodelo, existem outras duas diferenças principais entre o EMF e o MDR da Sun, conforme mostra a Figura 6. A primeira delas diz respeito à natureza de cada abordagem. Com a proposta de ser um repositório de metadados, o MDR oferece diferentes formas de armazenamento, tais como a representação em memória, banco de dados ou no sistema de arquivos, em estrutura de árvore B. Já o EMF tem foco na manipulação eficiente dos metamodelos, permitindo somente a representação em memória. Outra diferença é que, no MDR, somente as interfaces de manipulação do metamodelo são geradas. O código que as implementa é um só para qualquer metamodelo. Estando embutido no MDR, esse código utiliza programação reflexiva para possibilitar o acesso às informações do metamodelo. Já no

[6] `http://www.eclipse.org/emf/`

EMF, todo o código de acesso ao metamodelo é gerado, outra razão pela qual ele é mais rápido do que o MDR.

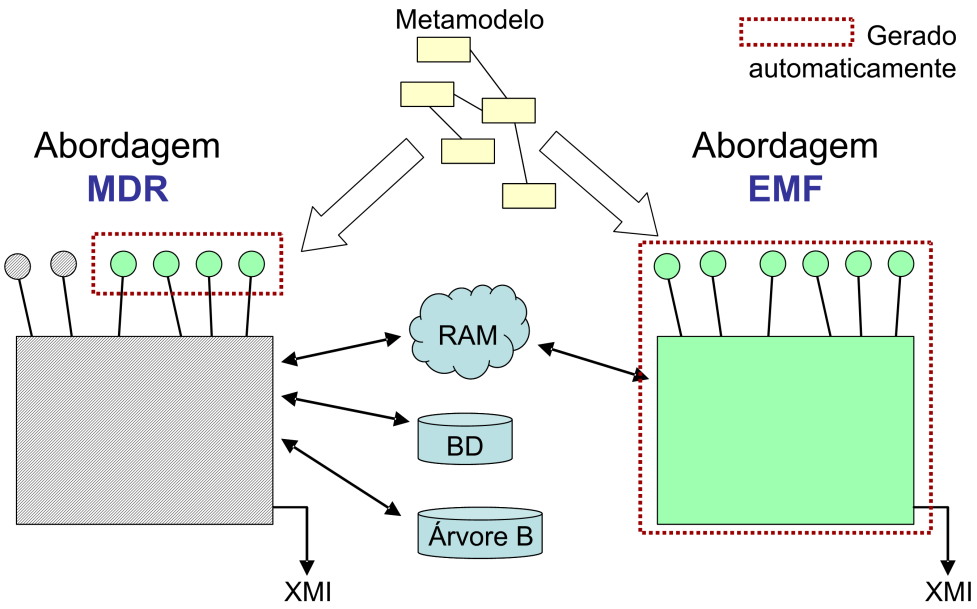


Figura 6: Comparação entre MDR e EMF

Outro projeto relacionado ao desenvolvimento orientado a modelos voltado à plataforma Eclipse é denominado GMT (*Generative Modeling Tools*[7]). Trata-se de uma iniciativa bastante recente, que se baseia nas idéias do padrões OMG, porém com algumas modificações.

Seu principal objetivo é servir de incubadora para projetos de ferramentas, linguagens e *frameworks* para dar suporte às tecnologias de transformação e geração de código. Dentre os sub-projetos do GMT, destacam-se:

- **ATL (*Atlas Transformation Language*[8])**: trata-se de uma linguagem para definição de transformações entre modelos. Originalmente projetada para ser compatível com o MOF, sua versão atual já é baseada no Ecore. É equivalente à linguagem QVT, do OMG, possuindo algumas diferenças (JOUAULT; KURTEV, 2006);
- **MOF Script**[9]: esse projeto envolve o desenvolvimento de ferramentas e *frameworks* para transformação de modelos para texto, com base em uma linguagem para definição dessas transformações; e
- **TCS - Textual Concrete Syntax**: esse projeto visa desenvolver ferramentas para a definição de DSLs textuais (JOUAULT; BÉZIVIN; KURTEV, 2006)[10].

[7] `http://www.eclipse.org/gmt/`
[8] `http://www.eclipse.org/gmt/atl/`
[9] `http://www.eclipse.org/gmt/mofscript/`
[10] DSLs serão abordadas mais adiante, no Capítulo 3.

Outro projeto, que fazia parte do GMT mas que se desvinculou por não mais se tratar de projeto incubado, é o **openArchitectureWare**[11]. Este projeto engloba ferramentas de modelagem textual, transformações de software e geração de código baseada em *templates*, além de outra funções pertinentes ao MDD.

O JET (*Java Emitter Templates*) (POPMA, 2004) consiste de um mecanismo de geração de código baseado em *templates* (CLEAVELAND, 1988). Através da inclusão de código Java dentro dos *templates*, permite a metaprogramação, ou seja, a criação de programas que criam programas. Qualquer comando Java pode ser utilizado, além de uma série de marcações (*tags*) que implementam comandos condicionais, de laços, formatação, entre outras funções úteis. O JET pode ser acoplado a arquivos XML ou modelos EMF, sendo portanto possível utilizá-lo como gerador de código para uma DSL.

Finalmente, um projeto que merece atenção nessa linha é o GMF (*Graphical Modeling Framework*[12]). Esse projeto permite a definição completa de um ambiente de modelagem para uma linguagem visual específica para um determinado domínio. A partir da definição do metamodelo da linguagem, da aparência gráfica dos elementos visuais dessa linguagem (notação), e das ferramentas necessárias para a criação dos elementos, é gerado um ambiente completo, que permite que o engenheiro de software construa modelos segundo a linguagem e notação definidos.

2.2.2.4 Outras abordagens

A *Vanderbilt University* abriga o projeto MIC (*Model Integrated Computing*), que pesquisa três elementos principais relacionados ao MDD: tecnologias para modelagem específica de domínio, um conjunto de ferramentas para modelagem, e um framework para análise formal, verificação e transformação de modelos. O principal produto desta pesquisa é o GME (*Generic Modeling Environment*)[13], um ambiente de metamodelagem que permite a criação de ferramentas de modelagem específica de domínio de forma simples e rápida.

A Microsoft, através do conceito de fábrica de software (GREENFIELD et al., 2004), também propõe uma abordagem para desenvolvimento de software baseada na reutilização sistemática e MDD. Esta abordagem busca imitar o funcionamento de uma fábrica tradicional, aplicando os conceitos no desenvolvimento de software. Ela define três elementos: (i) o esquema da fábrica de software, que descreve o que é necessário para construir os produtos da fábrica, (ii) o *template* da fábrica, que é uma instância do esquema, e (iii) um ambiente extensível, consistindo

[11] `http://www.openarchitectureware.org`
[12] `http://www.eclipse.org/gmf/`
[13] `http://www.isis.vanderbilt.edu/projects/gme/`

de ferramentas utilizadas para a produção, configuradas através do *template* da fábrica. As ferramentas desta abordagem estão focadas no *Microsoft Visual Studio*, principalmente com as *DSL Tools* (COOK et al., 2007), um conjunto de ferramentas para definição de linguagens específicas de domínio, geração de código, entre outras funções.

O UMT-QVT[14] [15] é uma ferramenta de código aberto que permite realizar transformações conforme exige o MDD. Um modelo serve de entrada para a ferramenta no formato XMI, que é então convertido para um formato intermediário. As transformações são baseadas em XSLT (W3C, 1999b), ou mesmo implementadas manualmente em linguagem de programação. O principal problema dessa abordagem está na dificuldade em se construir transformadores dessa forma, principalmente para transformar modelos. Neste sentido, as abordagens que utilizam linguagens visuais para definição de transformadores, como QVT e ATL, são mais apropriadas.

O MTF (*Model Transformation Framework*[16]), da IBM, contém um conjunto de ferramentas que auxiliam desenvolvedores em comparações, validações e criação de transformações entre modelos EMF. Também mantém um registro dos elementos transformados, permitindo rastreamento entre elementos gerados e a geração de relatórios para o usuário. É baseado em uma linguagem para definição dos mapeamentos entre os modelos, em um mecanismo de transformação capaz de interpretar esses mapeamentos, em um editor para as transformações, e no suporte para geração de texto a partir de modelos.

Finalmente, o projeto *openMDX*[17] foca na implementação dos conceitos da MDA, ou seja, o modelo é a implementação. Porém, a inserção de informações específicas da plataforma de execução é deixada para a etapa de implantação somente, não ocorrendo na etapa de projeto, como previsto pela MDA. Segundo os autores, isto traz uma série de benefícios, tais como a facilidade de portar modelos independentes de plataforma, e evita a necessidade de modelagem específica de plataforma, complexa e passível de erros, além de tornar a lógica da aplicação realmente independente da plataforma.

Todas estas abordagens e ferramentas têm alto potencial para auxiliar no MDD, mas da mesma maneira que no caso da reutilização, o papel do processo é fundamental para que as tentativas não sejam fadadas ao fracasso. Este é o assunto da próxima seção.

[14] Apesar de constar no nome desse projeto, essa ferramenta não é baseada na linguagem QVT do OMG.
[15] `http://umt-qvt.sourceforge.net`
[16] `http://www.alphaworks.ibm.com/tech/mtf`
[17] `http://www.openmdx.org/index.html`

2.2.3 O processo de desenvolvimento orientado a modelos

Não existe um consenso com relação às atividades de um processo de desenvolvimento orientado a modelos. O mais próximo disto é um modelo de maturidade denominado *MDD Maturity Model* (MODELWARE, 2006d). Este modelo foi definido com base na experiência das diversas empresas e instituições de pesquisa envolvidas com o *ModelWare*, uma iniciativa na área do MDD, descrita de forma mais detalhada no Capítulo 9. Esse modelo define as principais práticas e elementos de processo relacionados ao MDD, classificados em cinco níveis, conforme mostra a Figura 7.

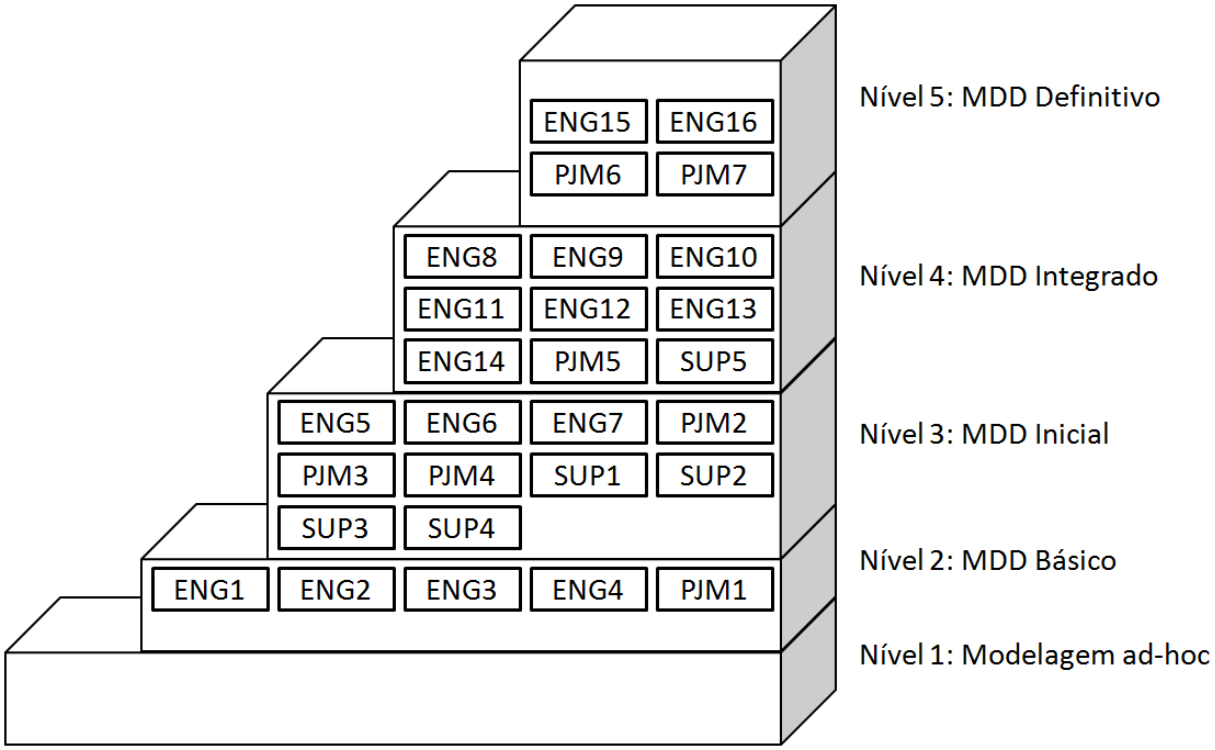


Figura 7: Modelo de maturidade em MDD (MODELWARE, 2006d)

Cada prática é identificada por um conjunto de caracteres que indica sua área e um número sequencial, sendo que ENG = Engenharia, PJM = Gerenciamento de projeto e SUP = Suporte. As práticas estão agrupadas nos cinco níveis conforme descrito a seguir:

- **Nível 1 - Modelagem *ad hoc***: neste nível, apenas o desenvolvimento tradicional é realizado. Práticas de modelagem são utilizadas apenas esporadicamente ou nunca;
- **Nível 2 - MDD Básico**: este nível é caracterizado pela utilização básica de modelos, cobrindo atividades simples do MDD, como a *decisão sobre as ferramentas e convenções de modelagem (ENG1 e PJM1)*. Modelos são utilizados apenas para guiar a implementação e documentação. Tipicamente, apenas *modelos técnicos (ENG2)* são

utilizados, incluindo todos os aspectos de um sistema, sem distinção entre aspectos de negócio ou aspectos específicos de plataforma. A *geração de código e de documentação (ENG3 e ENG4)* é feita diretamente a partir do modelo técnico.

- **Nível 3 - MDD Inicial**: neste nível introduz-se uma separação entre modelos de *negócio independentes de plataforma (ENG5)* e modelos específicos de plataforma. O objetivo é manter os aspectos de implementação independentes dos aspectos de negócio, de modo a melhorar a eficiência do processo de desenvolvimento. Neste nível, as práticas e artefatos do MDD são institucionalizadas, incluindo *o desenvolvimento de transformações modelo-para-texto (ENG6)* e a *verificação de modelos (ENG7)*. Na área de gerenciamento, este nível prevê atividades para *modelagem e aplicação do processo de MDD no projeto (PJM2 e PJM3)*, assim como a *definição, coleta e análise de métricas do projeto (PJM4)*. Neste nível também existe a preocupação com a *padronização das ferramentas e convenções de modelagem, dos procedimentos para coleta e análise das métricas e o estabelecimento de um repositório de modelos (SUP1, SUP2, SUP3 e SUP4)*;

- **Nível 4 - MDD Integrado**: o nível 4 é caracterizado por uma melhor integração entre os níveis de abstração de modelagem. *Metamodelos independentes e específicos de plataforma (ENG8 e ENG9)*, *modelos de negócio (ENG10)* e *transformações modelo-para-modelo (ENG11)* são definidos neste nível. Aqui também aparece a preocupação com a *rastreabilidade entre modelos (ENG12)*, com as *famílias de produtos (ENG13)*, caso seja este o foco da organização e com a *simulação de modelos (ENG14)*, visando detectar erros de maneira precoce. A modelagem do processo nesse nível passa a incluir os *processos automáticos do MDD (PJM5). Limites de desempenho de modelagem da organização são estabelecidos (SUP5)*, visando adequar os esforços de acordo com as características de cada projeto;

- **Nível 5 - MDD Definitivo**: neste último nível, todo o conhecimento da organização é mantido na forma de modelos e transformações, que são o foco do processo de desenvolvimento. Práticas para o *desenvolvimento de linguagens específicas de domínio (ENG15)* e a *verificação e validação de produtos (ENG16)* são complementadas com práticas para *estabelecer e manter artefatos de modelagem de software estratégicos para o MDD (PJM6)* e *promulgar o modelo de processo do projeto (PJM8)*, tornando o desenvolvimento mais controlável;

Algumas dessas práticas estão intimamente relacionadas com a abordagem desta tese, conforme descrito no Capítulo 4.

2.3 Considerações finais

Neste capítulo foram discutidos os principais conceitos e técnicas da reutilização de software e do desenvolvimento orientado a modelos. Enquanto as técnicas para reutilização buscam aproveitar ao máximo o que já existe em termos de artefatos de software produzidos anteriormente, o desenvolvimento orientado a modelos busca elevar o nível de abstração no qual o desenvolvedor trabalha, escondendo detalhes da plataforma de implementação e empregando as habilidades de automação do computador para ajudar nas tarefas de tradução entre problema e solução e nas tarefas repetitivas.

Estas técnicas formam um conjunto de ferramentas com potencial para fazer com que os objetivos da reutilização e do MDD sejam alcançados. Porém, sem a existência de um processo bem definido, que dê suporte à institucionalização de práticas comprovadas e sistemáticas, este potencial pode não atingir seu ponto máximo, e como resultado os benefícios associados à reutilização e ao MDD podem se perder durante as tentativas. Assim, o papel do processo é o de coordenar esforços de maneira que o sucesso passa a depender de fatores mais controláveis, como planejamento, gerenciamento e treinamento, ao invés de ser o resultado de talento e experiências individuais isoladas. A literatura apresenta importantes contribuições nas áreas de processos para reutilização de software e MDD, porém a combinação de ambas permanece relativamente inexplorada.

Existem alguns pontos de intersecção, com impacto significativo no processo, que precisam ser devidamente analisados antes de se tentar uma combinação entre reutilização e MDD. No próximo capítulo esta intersecção é analisada de forma mais aprofundada, buscando identificar como as características de cada abordagem pode influenciar a outra de forma a proporcionar um melhor resultado final.

3 *Reutilização de software e desenvolvimento orientado a modelos*

Reutilização de software e desenvolvimento orientado a modelos são duas áreas distintas, mas com muitos objetivos em comum. Ambas procuram mais qualidade e produtividade no desenvolvimento de software por meio da redução de esforço repetitivo e da adoção de soluções que agregam conhecimento prévio. No caso da reutilização, busca-se encapsular, em forma de artefatos de software diversos, informações e conceitos reutilizáveis de um domínio, incluindo algoritmos, estruturas de dados, funções, etc. No caso do desenvolvimento orientado a modelos, busca-se encapsular também o conhecimento necessário para se produzir esses artefatos, em forma de transformações que mapeiam conceitos de mais alto nível até o código.

É particularmente interessante o fato de que o desenvolvimento orientado a modelos pode promover a reutilização em mais alto nível, conforme sempre defendido e idealizado por inúmeros autores (NEIGHBORS, 1980; KRUEGER, 1992; GRISS, 1995; FRAKES; ISODA, 1994; JACOBSON; GRISS; JONSSON, 1997). Este fato evidencia o potencial que o MDD possui para elevar os níveis de reutilização.

Nesta tese, foram identificados dois pontos importantes de intersecção entre a reutilização e o MDD: (i) a análise de escopo, comunalidade e variabilidade (análise SCV); e (ii) a implementação da variabilidade. Estes pontos tomam formas distintas em cada abordagem, conforme descreve-se a seguir.

3.1 Análise SCV

Uma das principais atividades necessárias para projetos de reutilização em grande escala é chamada de análise SCV (*Scope, commonality, and variability* ou escopo, comunalidade e variabilidade). Ela oferece aos engenheiros de software uma maneira sistemática de pensar sobre e identificar a família de produtos que estão criando, ajudando, entre outras coisas (COPLIEN; HOFFMAN; WEISS, 1998):

- Na criação de um projeto que contribui para a reutilização e facilita as mudanças;
- Na previsão de como um projeto pode falhar ou ter sucesso, à medida que evolui; e
- Na identificação de oportunidades para automatizar a criação dos membros da família.

Em outras palavras, identificando-se os pontos comuns e variáveis de um domínio, pode-se construir artefatos reutilizáveis que representam os pontos comuns, e projetar mecanismos configuráveis ou adaptáveis para representar os pontos variáveis (LEE; KANG, 2004). Desta forma, a reutilização é antecipada e pode ser efetivamente incluída como requisito durante a análise, projeto e implementação do domínio.

A SCV é realizada em ambas as áreas, de reutilização de software e desenvolvimento orientado a modelos, conforme apresenta-se nas seções a seguir.

3.1.1 SCV na reutilização de software

Na área de reutilização de software, a modelagem de *features*[1] (KANG et al., 1990; LEE; KANG; LEE, 2002) é amplamente utilizada como mecanismo de representação da variabilidade (LEE; MUTHIG, 2006).

Features são quaisquer conceitos ou características distintas e proeminentes de um domínio, e que são externamente visíveis a um usuário ou outros *stakeholders* (por exemplo, analistas, projetistas, desenvolvedores, etc) (LEE; KANG; LEE, 2002). Trata-se de um conceito simples e orientado ao domínio do problema, e por isso a modelagem de *features* constitui um meio natural de comunicação entre os diferentes *stakeholders*, sendo intuitivo para as pessoas expressarem a comunalidade e variabilidade de um domínio por meio de *features* (LEE; MUTHIG, 2006).

A modelagem de *features* identifica os pontos comuns e variáveis de um domínio em termos das *features*. Esta técnica está descrita da seguinte forma por Kang et al. (1998):

- *Features* são identificadas e classificadas em termos de *features* de capacitação, de tecnologia do domínio, de técnicas de implementação e de ambientes de operação. *Features* de capacitação são características visíveis ao usuário que podem ser identificadas como serviços, operações e características não-funcionais. *Features* de tecnologia do

[1] Nesta tese é utilizado o termo original em inglês "*feature*" para se referir a uma característica de um domínio conforme a técnica proposta originalmente por Kang et al., ao invés de sua tradução para o português, pois o termo em inglês remete de forma mais direta e menos ambígua à técnica, enquanto o termo em português pode ser utilizado em outros contextos.

domínio representam maneiras para se implementar os serviços ou operações. *Features* de técnicas de implementação são funções ou técnicas genéricas utilizadas para implementar serviços, operações e funções do domínio. *Features* de ambiente de operação representam o ambiente no qual as aplicações são executadas;

- *Features* comuns a todos os produtos do domínio são modeladas como mandatórias, enquanto *features* que diferem entre os produtos podem ser opcionais ou alternativas. *Features* opcionais representam *features* selecionáveis dentro do domínio, e *features* alternativas indicam que apenas uma *feature* pode ser selecionada para um produto; e
- Um diagrama de *features* é uma hierarquia gráfica que captura relações estruturais e conceituais entre as *features*. Há três tipos de relações: composição, generalização e implementação. Regras adicionais complementam o modelo com relações de dependência ou exclusão mútua entre as *features*.

Este modelo é fundamentado na presença ou ausência das *features*, e é capaz de cobrir grande parte da variabilidade presente na maioria dos domínios. Porém, em alguns casos é necessário maior poder expressivo e detalhes que não podem ser expressos através destas regras.

Por este motivo, Czarnecki, Helsen e Eisenecker (2004b) apresentam algumas extensões propostas na literatura para o modelo de *features*, visando dar mais flexibilidade e capacidade para representar uma maior variedade de casos. As seguintes extensões foram propostas:

- **Cardinalidade de *features***: *features* podem ser anotadas com cardinalidades, como [1..*] ou [3..3]. Desta forma, *features* mandatórias e opcionais podem ser consideradas casos especiais das cardinalidades [1..1] e [0..1], respectivamente;
- **Grupos e cardinalidade de grupos**: *features* alternativas podem ser vistas como um mecanismo de agrupamento. Esta extensão propõe o uso de cardinalidades também nos grupos. Por exemplo, um grupo de alternativas, onde pelo menos uma e no máximo uma *feature* deve ser selecionada, é anotado com a cardinalidade [1..1]. Em outro exemplo, caso entre duas ou quatro opções do grupo devam ser selecionadas, a cardinalidade utilizada é [2..4];
- **Atributos**: em alguns casos a relação de uma *feature* com uma *subfeature* é tão simples que o uso de atributos associados a um tipo (como inteiro ou caractere) é mais elegante e simplifica o modelo;
- **Relacionamentos**: diferentes relacionamentos podem ser utilizados para relacionar *features*, tais como parte-de ou generalização;

- **Categorias de *features* e anotações**: existem diversas categorias de *features* que estendem aquelas propostas originalmente por Kang et al. (1990), incluindo: *features* funcionais, arquiteturais, entre outras; e

- **Modularização**: um diagrama de *features* pode conter um ou mais nós especiais, cada um representando um diagrama de *features* separado. Este mecanismo permite a quebra de diagramas grandes em diagramas menores, assim como a reutilização de partes comuns em diferentes locais.

Existem diferentes notações propostas para a modelagem de *features*, com algumas diferenças em relação à notação original proposta por Kang et al. (1990). Porém, uma característica desta técnica é que a notação é fixa, ou seja, pode não ser a forma mais intuitiva para representar os conceitos de um domínio. Por este motivo, no MDD utiliza-se uma maneira mais flexível para a análise SCV, conforme descrito a seguir.

3.1.2 SCV no desenvolvimento orientado a modelos

Uma das formas para se realizar a análise SCV no desenvolvimento orientado a modelos envolve a utilização de linguagens específicas de domínio (DSL - *Domain-Specific Language*) para representação da variabilidade do domínio (VISSER, 2007). Uma DSL pode ser tão simples como um *wizard*, ou uma linguagem completa, desenvolvida especialmente para representar a variabilidade (CZARNECKI, 2005).

Uma DSL é uma linguagem pequena, normalmente declarativa, focada em um domínio de problema em particular (DEURSEN; KLINT; VISSER, 2000). DSLs existem há um longo tempo. Mernik, Heering e Sloane (2005) citam os exemplos da linguagem APT para controle numérico, que data de 1957-58, e da mais famosa BNF, ou *Backus-Naur Form*, linguagem para especificação de gramáticas criada em 1959. Desde então, diversas linguagens vêm sendo desenvolvidas e utilizadas, e a literatura nesta área é bastante rica (DEURSEN; KLINT; VISSER, 2000; MERNIK; HEERING; SLOANE, 2005)[2].

Uma linguagem específica de domínio pode ser textual (permitindo especificar programas) ou visual (permitindo especificar modelos ou diagramas), e normalmente possui três elementos: a sintaxe abstrata, a sintaxe concreta e a semântica.

A **sintaxe abstrata** define os conceitos do domínio, e as relações e restrições que se aplicam a estes conceitos. Em linguagens textuais, é representada por uma árvore (chamada

[2]Uma discussão mais aprofundada sobre DSLs e seu papel na reutilização é apresentada no Apêndice A.

de AST ou *Abstract Syntax Tree*), que armazena as palavras do vocabulário da linguagem e seu agrupamento válido em forma de sentenças. Em linguagens de modelagem, corresponde ao metamodelo que define a estrutura dos modelos que podem ser criados (GUIZZARDI; PIRES; SINDEREN, 2002). Uma vez que este metamodelo é uma especificação de uma conceitualização do domínio, pode ser considerado uma *ontologia* (KURTEV et al., 2006).

A **sintaxe concreta** provê um sistema para representar os conceitos do domínio de forma concreta. Consiste de símbolos, que podem ser caracteres organizados em palavras segundo uma gramática bem definida (linguagem textual), ou ícones gráficos com características visuais que representam diferentes atributos, como cor, posição, tamanho e forma (linguagem visual) (GUIZZARDI; PIRES; SINDEREN, 2002). Trata-se de uma representação superficial, que pode ser diferente da representação interna que é obtida com a sintaxe abstrata. De fato, uma DSL pode possuir múltiplas sintaxes concretas (KLEPPE, 2007).

A **semântica** define o significado dos elementos da sintaxe abstrata, e pode variar de acordo com o objetivo desejado. Por exemplo, se o objetivo da linguagem é facilitar a comunicação interpessoal, a semântica define o que cada frase ou modelo significa para um leitor humano. Neste sentido, pode-se considerar a semântica como sendo um elemento passível de diversas interpretações, uma vez que qualquer pessoa pode atribuir um significado a determinada palavra ou ícone. No entanto, no contexto do MDD, a semântica é definida em forma de ações a serem executadas por um interpretador automático. As abordagens para o tratamento da semântica no MDD podem se enquadrar em ao menos quatro categorias (KLEPPE, 2007):

1. **Denotativa**: descrições matemáticas que representam o significado do programa ou modelo;
2. **Operacional**: também conhecida como execução canônica (CLEENEWERCK; KURTEV, 2007), consiste na interpretação do programa ou modelo como uma sequência de passos a serem executados. Normalmente resulta em uma máquina de estados;
3. **Tradutiva**: consiste na tradução do programa ou modelo em outra linguagem conhecida; e
4. **Pragmática**: uma ferramenta, normalmente conhecida como *implementação de referência*, executa o programa ou modelo.

A abordagem tradutiva é uma das mais indicadas (KLEPPE, 2007), e também uma das mais empregadas. Por exemplo, em um gerenciador de interfaces visuais, como aqueles presentes em IDEs como NetBeans ou Visual Studio, as interfaces são especificadas de forma visual, em uma

linguagem específica para o domínio GUI (*Graphical User Interface* ou Interface Gráfica com o Usuário). Nesta linguagem, especifica-se os atributos de cada componente, como posição e tamanho, assim como os eventos e ações associados. Como resultado, é gerado código que traduz a semântica da linguagem (posição, tamanho, atributos, eventos de cada componente) em uma GPL (*General Purpose Language* ou Linguagem de Propósito Geral, como por exemplo UML, Java, C#, C++, VB, entre outras).

A definição da linguagem normalmente requer um metamodelo que seja capaz de capturar os pontos comuns e variáveis do domínio, e não a mais comumente utilizada BNF. Um metamodelo é uma estrutura similar a um diagrama de classes, e possui elementos como classes, atributos, associações e agregações. Seja qual for a representação visual final (diagrama visual ou representação textual), a existência de um metamodelo como esquema conceitual em geral indica que uma maior quantidade de instâncias pode ser expressa. Isto ocorre pelo simples fato de que regras gramaticais BNF descrevem uma árvore, enquanto um metamodelo descreve um grafo, e árvores são subconjuntos de grafos (KLEPPE, 2007).

Com relação à sintaxe concreta, ambas as possibilidades (linguagens visuais ou textuais) devem ser consideradas, e a escolha depende de uma análise mais aprofundada (PETRE, 1995). Entre os fatores a serem analisados, pode-se citar o fato de que a implementação de interpretadores textuais (*parsers*) é uma tarefa reconhecidamente difícil (FEILKAS, 2006; CLEAVELAND, 1988). Além disso, é impossível expressar todas as informações necessárias em gramáticas livres de contexto (FEILKAS, 2006). Linguagens visuais também são mais intuitivas (ESSER; JANNECK, 2001), e portanto resultam em maior facilidade de utilização por especialistas não experientes com tecnologia da informação. Muitas vezes, porém, uma combinação de múltiplos formatos de entrada, ou mesmo múltiplas linguagens, é necessária (WILE, 2004; TOLVANEN; KELLY, 2001).

As extensões do modelo de *features* descritas na seção anterior se aproximam bastante da capacidade de representação de variabilidade que pode ser alcançada através de uma DSL. A diferença é que com uma DSL tem-se um maior poder expressivo, pois o foco na linguagem visa oferecer uma sintaxe concreta que seja familiar e intuitiva para os especialistas do domínio, enquanto a modelagem estendida de *features* ainda está atrelada a uma notação fixa.

3.2 Implementação da variabilidade

Conforme já identificado por Parnas (1976) há décadas atrás, além de muitos outros desde então (MILI; MILI; MILI, 1995; EZRAN; MORISIO; TULLY, 2002), sistemas de software raramente

são compostos exclusivamente por elementos novos. Muitos compartilham uma base comum, diferindo apenas em pontos isolados. Estes expressam as variações, ou variabilidade de software.

Para alguns autores, como Svahnberg, Gurp e Bosch (2005), o termo variabilidade de software não significa apenas as diferenças entre sistemas de software, mas também sua capacidade em ser estendido, modificado, customizado ou configurado de forma eficiente para uso em um contexto particular. Trata-se portanto de um atributo de qualidade, assim como manutenibilidade ou usabilidade. Nesta tese, porém, utiliza-se o termo variabilidade como referência às possíveis variações de um conjunto de produtos de software.

Para garantir que um sistema possua um suporte adequado à variabilidade, ela deve ser gerenciada durante todo o seu ciclo de vida, desde a análise até o projeto e implementação. A análise SCV, descrita na seção anterior, corresponde à identificação e modelagem da variabilidade. Em seguida, as atividades de projeto e implementação devem cuidar para que esta variabilidade identificada seja devidamente implementada, em forma de mecanismos configuráveis e adaptáveis (LEE; KANG; LEE, 2002).

O suporte à variabilidade ocorre de forma distinta nas áreas de reutilização e MDD.

3.2.1 Implementação da variabilidade na reutilização de software

Existem inúmeras maneiras de se implementar variabilidade. Normalmente, a literatura na área de reutilização apresenta soluções baseadas em código-fonte. Neste contexto, dentre as principais técnicas para implementação da variabilidade, destacam-se as seguintes, conforme citam Matos Jr (2008), Anastasopoulos e Gacek (2001), Muthig e Patzke (2003):

- **Compilação condicional**: consiste na marcação de código com diretrizes de pré-processamento, para incluir ou excluir trechos de código de acordo com a presença ou ausência de uma variante. É um mecanismo simples, porém muito poderoso e robusto, presente em grande parte das linguagens de programação atuais;

- **Herança**: é um mecanismo de linguagens orientadas a objetos, consistindo na criação de um tipo abstrato que representa o ponto de variação, e diversos tipos concretos, cada um sendo uma alternativa. Porém, utilizando-se herança simples, somente uma alternativa pode estar presente em um determinado produto. Nestes casos, o uso de *Mixins*, herança múltipla ou herança parametrizada pode ajudar;

- **Agregação/delegação**: esta técnica consiste em repassar requisições entre objetos,

caso um deles não seja capaz de atender a uma requisição. A variabilidade pode ser implementada desta forma, incluindo uma funcionalidade padrão ou mandatória em um objeto, e uma funcionalidade variante em outro objeto, a ser delegado. Esta técnica funciona com *features* opcionais, mas apresenta problemas para alternativas, já que é necessário decidir para quais objetos deve ser feita a delegação;

- **Parametrização**: consiste na inserção de parâmetros em componentes e interfaces, para seleção das variantes. O componente é então responsável por executar um comportamento diferente de acordo com o parâmetro especificado;

- **Programação orientada a aspectos (AOP - *Aspect-Oriented Programming*)**: com AOP (KICZALES et al., 1997), funcionalidades mandatórias podem ser implementadas de modo padrão, enquanto funcionalidades alternativas são implementadas em forma de aspectos, a serem combinados posteriormente, num processo conhecido como *aspect weaving*. Ou seja, antes da execução, um programa instancia uma configuração do seu código executável selecionando as alternativas de aspectos que sejam adequadas ao seu estado atual;

- **Arquivos de configuração**: consiste na criação de arquivos separados que contêm as informações variantes, a serem selecionadas e incluídas no produto;

- **Carregamento dinâmico de classes**: permite que um produto solicite uma classe dinamicamente, em tempo de execução. Esta classe é então carregada na memória e executada. Esta técnica permite a inclusão dinâmica de alternativas, de acordo, por exemplo, com parâmetros ou arquivos de configuração.

Outra técnica também bastante utilizada no suporte à variabilidade é o uso de padrões de projeto (KEEPENCE; MANNION, 1999; ANASTASOPOULOS; GACEK, 2001; LEE; KANG, 2004; ALMEIDA et al., 2007b). Padrões como o *Abstract Factory*, *Singleton*, *Factory Method*, *Prototype*, *Strategy*, *Template Method*, *Builder*, *Director*, *Observer*, *Decorator*, *Composite*, *Adapter*, *Bridge*, *Chain of Responsibility* e *Command* (GAMMA et al., 1995), entre outros, podem potencializar o uso das técnicas para implementação da variabilidade descritas.

Finalmente, pode-se implementar variabilidade utilizando técnicas de gerência de configuração (MUTHIG et al., 2002). Nesta abordagem, variantes alternativas de um mesmo artefato são incluídas ou excluídas através de mecanismos de controle de versões, selecionando-se a versão que possui a variante desejada, por exemplo. Esta abordagem é utilizada por muitos profissionais, mas frequentemente leva a problemas graves quando as variações são muito complexas (MUTHIG; PATZKE, 2004). Isto porque, em geral, dependem de

um profundo conhecimento do sistema em diferentes níveis de detalhes, por parte do engenheiro de software.

3.2.2 Implementação da variabilidade no desenvolvimento orientado a modelos

No desenvolvimento orientado a modelos, a geração de código é normalmente empregada com o intuito de se implementar a variabilidade do domínio. Utilizando esta técnica, as tarefas repetitivas relacionadas à implementação da variabilidade são automatizadas, e o desenvolvedor pode obter soluções completas através de especificações em alto nível, normalmente uma DSL oriunda da análise SCV (Seção 3.1.2).

Um gerador pode ser aplicado em conjunto com as técnicas descritas na seção anterior. A diferença é que um gerador possui um grau a mais de liberdade, podendo introduzir variações também no momento da geração. Com isto, a granularidade das partes variantes está abaixo do nível de componentes, ou seja, o próprio código dentro do componente pode ser feito genérico. Isto é alcançado através de fragmentos de código variantes que podem ser sistematicamente arranjados para produzir a implementação de um componente (MUTHIG; PATZKE, 2004).

Para construir geradores, normalmente o uso de *templates* é preferido. Um *template* é um arquivo de texto qualquer instrumentado com construções de seleção e expansão de código (CZARNECKI; EISENECKER, 2000). Estas construções são responsáveis por realizar consultas em uma entrada, que pode ser um programa ou especificação textual, um conjunto de janelas e diálogos, no estilo *wizard*, ou mesmo diagramas (CLEAVELAND, 1988). A informação resultante destas consultas é então utilizada como parâmetro para produzir código personalizado, em qualquer linguagem textual.

Para ilustrar este processo, considere o seguinte cenário: uma empresa deseja agilizar a produção de formulários HTML para incluir em sua aplicação Web. Existem centenas de formulários distintos, e os campos de cada formulário mudam constantemente, de forma que a manutenção deste conjunto de páginas é normalmente trabalhosa.

Uma solução baseada em DSL e *templates* é composta de pelo menos três elementos principais. O **primeiro** é um formato de entrada, que permite que um desenvolvedor especifique o conteúdo de cada formulário, tais como os campos, tipo de dados, tamanho, etc. Dentre os possíveis formatos, o XML apresenta vantagens pela facilidade de se processar as informações. Bibliotecas com suporte a linguagens como XPATH (W3C, 1999a) facilitam o trabalho de se consultar as informações.

O **segundo** elemento é o *template*, neste caso uma página HTML anotada, segundo o formato JET, com código JAVA que consulta a entrada do gerador. O **terceiro** elemento é o processador de *templates*, responsável por instanciar o *template* com base na entrada. A Figura 8 ilustra esse processo, onde cada trecho do *template* é processado, por meio do JET (Ver Seção 2.2.2.3), para consultar o arquivo de entrada e produzir o código correspondente.

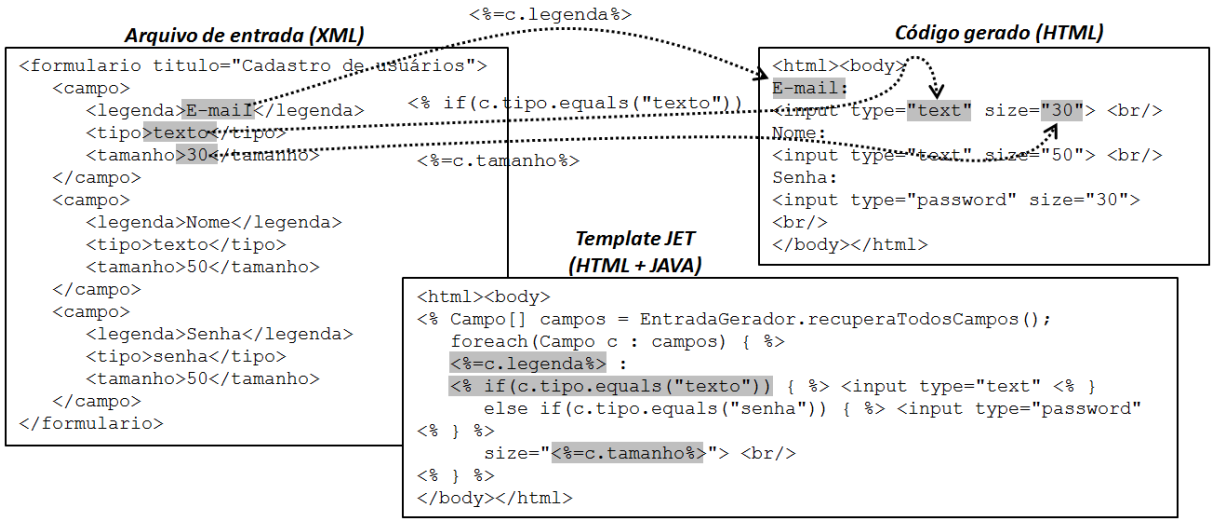


Figura 8: Geração de código baseada em *templates*

Apesar de simples, este exemplo demonstra os possíveis benefícios da abordagem. A manutenção é facilitada, pois não é necessário inspecionar trechos obscuros de código HTML para realizar mudanças. É possível produzir novos formulários simplesmente adicionando um novo elemento `<formulario>` na especificação, ao invés de copiar um formulário existente e modificar o código.

Destaca-se também a facilidade de adoção por parte de desenvolvedores, uma vez que esta abordagem permite gerar código em qualquer linguagem ou formato (CZARNECKI; EISENECKER, 2000). Além disso, enquanto outras abordagens produzem código-fonte que normalmente não seria escrito à mão (CLEAVELAND, 1988), o uso de *templates* propicia a geração de código mais legível, o que é desejável (VÖLTER, 2008) ou mesmo crítico em muitos casos (CZARNECKI; EISENECKER, 2000). Isto porque o *template* é parecido com a saída desejada, acrescido de anotações e instruções que realizam a consulta na entrada e produzem a saída desejada (CLEAVELAND, 2001).

Existem também padrões de projeto que podem ser utilizados com geradores de código. A exemplo dos padrões de projeto de software tradicionais, estes buscam oferecer soluções para problemas comuns envolvendo o desenvolvimento de software baseado em geração de código. Völter (2003), Völter e Bettin (2004) apresentam uma lista com estes padrões.

3.3 Considerações finais

Neste capítulo foram apresentados detalhes sobre algumas características importantes relacionadas à reutilização de software e ao desenvolvimento orientado a modelos que têm sido adotadas de maneira prática, em função dos conceitos e objetivos inerentes a cada uma dessas duas abordagens. De fato, cada área possui objetivos em comum, mas emprega técnicas distintas, sendo necessário cuidado especial ao combiná-las em uma única abordagem.

Em especial, destaca-se o fato de que a combinação de reutilização e MDD não ocorre somente em pontos isolados do processo, mas deve ser iniciada na análise, passando pelo projeto, até a implementação, e deve incluir um gerenciamento adequado da variabilidade. Nesta tese, a engenharia de domínio foi identificada como a estratégia de processo a ser utilizada e complementada com técnicas do MDD, de forma que a preocupação com a modelagem se faz presente em todo o processo de reutilização, efetivamente elevando o nível de abstração do desenvolvimento.

Nos próximos capítulos são apresentadas uma visão geral e descrições detalhadas sobre as atividades da abordagem definida nesta tese, com especial atenção em como os pontos discutidos neste capítulo são combinados de forma a oferecer um melhor suporte à reutilização em alto nível, utilizando MDD.

4 Visão geral da abordagem

Neste capítulo é apresentada uma visão geral da abordagem orientada a modelos para reutilização de software proposta nesta tese. São apresentados o seu objetivo, sua estrutura e uma breve descrição de suas atividades. No final é feita uma análise com os modelos de maturidade de reutilização e MDD, visando melhor ilustrar seu escopo e cobertura no processo de software.

4.1 Objetivo da abordagem

A abordagem orientada a modelos para reutilização de software tem como objetivo:

Reutilizar... Sendo uma abordagem para reutilização de software, o objetivo é possibilitar que o desenvolvimento de software possa reutilizar artefatos pré-existentes ao invés de construir tudo a partir do início (KRUEGER, 1992). Idealmente, nenhuma oportunidade de reutilização deveria ser perdida, ou seja, deve-se tentar reaproveitar o máximo possível de software existente durante um novo desenvolvimento;

De forma sistemática... Esta reutilização, em contraste com uma abordagem *ad hoc*, que é altamente dependente de iniciativa, conhecimento e competência individual, deve ser sistemática. Deve se basear em conceitos da engenharia, de forma a ser um processo controlável, gerenciável e repetível (ROMBACH, 2000);

Considerando-se uma família de sistemas similares... Para alcançar esta reutilização sistemática, além do foco no processo (ENDRES, 1993; BAUER, 1993; GRISS, 1994, 1995; JOOS, 1994), deve-se promover uma mudança de paradigma, do desenvolvimento de sistemas únicos para a preocupação com uma família de sistemas similares (FRAKES; ISODA, 1994). Dessa forma, a reutilização ocorre de forma mais ampla, aproveitando o potencial de similaridade entre as aplicações (EZRAN; MORISIO; TULLY, 2002; MILI; MILI; MILI, 1995);

Centrada em um domínio e seus subdomínios... Um domínio compreende um conjunto de aplicações similares, que atendem a uma determinada área de problema do mundo real (NEIGHBORS, 1980). Desta forma, focando-se em um domínio, tem-se um escopo bem definido para se planejar a reutilização. O desenvolvimento centrado no domínio inclui a identificação das características comuns do domínio, a criação de modelos do domínio, o projeto arquitetural centrado no domínio, e a implementação de artefatos que sejam reutilizáveis no contexto do domínio. Além disso, um domínio pode conter subdomínios distintos, cada um com maior ou menor potencial de automação. A abordagem busca identificar esses subdomínios e prover meios para gerenciar seu ciclo de vida e sua integração, considerando a arquitetura final composta por diferentes artefatos de software, tais como modelos, componentes, transformações e geradores de código;

Com foco em aplicações concretas... Identificar o número e o tamanho dos sistemas que fazem parte do domínio é uma tarefa que, se realizada sem critério, pode levar a alguns problemas. Uma análise livre tende a incluir diversos requisitos e sistemas que, apesar de serem relacionados e pertinentes ao domínio em questão, podem nunca fazer parte de uma aplicação desenvolvida pela organização. Assim, o escopo do domínio deve ser restrito a aplicações e produtos de software concretos, e não idéias e conceitos gerais do domínio. A análise deve focar em aplicações existentes, futuras e potenciais (CLEMENTS; NORTHROP, 2002; ALMEIDA et al., 2007b), buscando coletar, organizar, armazenar e analisar as experiências em sua construção, conforme prevê a engenharia de domínio (CZARNECKI; EISENECKER, 2000);

Elevando a importância dos modelos... Os artefatos reutilizáveis desenvolvidos não devem se restringir a código-fonte apenas, uma vez que o desenvolvimento centrado no código-fonte apresenta diversos problemas, como discutido na Seção 2.2. Assim, a abordagem deve elevar o nível de importância dos modelos no processo (MELLOR; CLARK; FUTAGAMI, 2003), de forma que possam não somente ser utilizados como mecanismos de comunicação entre as equipes e guias para o desenvolvimento de software, mas também sirvam como entradas para mecanismos automatizados de transformação de software e geração de código (SCHMIDT, 2006); e

Encapsulando o conhecimento em diferentes formas... Como resultado, a infraestrutura de reutilização que é produzida pela abordagem inclui não somente componentes de código, mas também artefatos do MDD:

1. **Linguagens e ferramentas de modelagem específicas de domínio:** consistem de linguagens com poder expressivo capaz de representar os conceitos do domínio, e

ferramentas capazes de criar modelos segundo estas linguagens, de forma a facilitar o entendimento de requisitos e a comunicação entre clientes/desenvolvedores; e

2. **Transformações de software:** consistem de mecanismos que automaticamente transformam um artefato de software em outro. Por exemplo, transformam modelos em código executável, *scripts* de banco de dados, arquivos de configuração, além de outros artefatos que fazem parte do produto final. As transformações podem ser do tipo modelo-para-texto, ou seja, transformações que geram código, ou modelo-para-modelo, que transformam um modelo em outro.

Em resumo, define-se o objetivo desta abordagem da seguinte forma.

***Reutilizar** software **de forma sistemática**, **considerando-se uma família de sistemas similares**, seguindo uma abordagem **centrada em um domínio e seus subdomínios**, **com foco em aplicações concretas**, em um processo que **eleva a importância dos modelos** e que **encapsula o conhecimento em diferentes formas**: componentes de software, linguagens e ferramentas específicas de domínio e transformações de software.*

4.2 Definição da abordagem

A abordagem foi definida após estudos da literatura e desenvolvimento de estudos de caso de caráter exploratório. Foram estudados artigos científicos e relatórios de empresas na área de reutilização e desenvolvimento orientado a modelos. Os resultados do estudo da área de reutilização foram publicados e podem ser encontrados na literatura (ALMEIDA et al., 2005; ALMEIDA, 2007), enquanto os resultados do estudo da área de desenvolvimento orientado a modelos encontram-se no Capítulo 9.

Cada fase da abordagem foi definida separadamente e seguindo sua ordem natural. Buscou-se incorporar os pontos fortes das outras abordagens da literatura, por meio de atividades já bem estabelecidas, como aquelas do projeto e avaliação arquiteturais desenvolvidas pelo SEI (*Software Engineering Institute*) (BASS; CLEMENTS; KAZMAN, 2003; CLEMENTS; KAZMAN; KLEIN, 2004). Buscou-se também preencher as lacunas e melhorar os pontos fracos das abordagens descritas na literatura, através de novas técnicas, como as atividades para identificação e gerenciamento dos subdomínios, descritas no Capítulo 5. Nos pontos onde a literatura apresenta soluções que não são diretamente aplicáveis ao contexto do MDD, o objetivo foi adaptar as técnicas existentes para o contexto desta tese. Por exemplo, no Capítulo 6, que descreve o uso de padrões de projeto normalmente utilizados para o gerenciamento da variabilidade (ALMEIDA et al., 2007b; KEEPENCE; MANNION, 1999; LEE; KANG, 2004), foram

incorporados detalhes referentes à sua utilização em conjunto com transformações e geradores de código. Da mesma forma, no Capítulo 7 são descritas diretrizes de apoio às atividades de implementação de DSLs e geradores no contexto da Engenharia de Domínio. Outros exemplos são descritos durante os próximos capítulos, no próprio texto que descreve a abordagem.

Durante a definição das fases, foram desenvolvidos estudos de caso de caráter exploratório visando testar e refinar cada atividade e artefato produzido. Estes estudos envolveram o domínio de aplicações Web e posteriormente evoluíram para o primeiro dos estudos de caso utilizados na avaliação, conforme descrito no Capítulo 8. Reuniões com o grupo de pesquisa e outros especialistas com os quais o autor tem colaborado também ajudaram a testar e refinar a abordagem.

4.3 Estrutura da abordagem

A abordagem está dividida em três fases, que seguem a divisão de fases da engenharia de domínio: análise, projeto e implementação. A **análise de domínio** tem como objetivo identificar o escopo do domínio, identificar pontos comuns e variáveis do domínio, identificar possíveis subdomínios automatizáveis, e produzir documentação sobre estas informações. O **projeto do domínio** tem como objetivo principal projetar uma arquitetura específica de domínio. Esta arquitetura deve contemplar as variabilidades identificadas durante a análise, e oferecer suporte ao seu gerenciamento, considerando a existência de diferentes tipos de variabilidades em cada subdomínio identificado e artefatos do MDD, como DSLs, ferramentas de modelagem e transformadores/geradores de código. Por fim, na **implementação do domínio**, são produzidos os artefatos de implementação, como componentes, DSLs, ferramentas de modelagem, transformações e geradores de código, de forma a implementar a variabilidade conforme prevista e planejada nas fases de análise e projeto.

A abordagem é apresentada por meio de diferentes elementos:

- **Atividades**: são os elementos principais da abordagem. Elas representam uma instância concreta de uma determinada prática, e definem de forma sistemática a sua execução. Para cada atividade, são identificados os papéis, entradas e saídas associados;

- **Produtos de trabalho**: são os artefatos produzidos ou utilizados durante a abordagem, tais como documentos, modelos, código, etc. Cada produto de trabalho pode ser modificado ou não por uma atividade, e portanto ele possui diferentes estados em diferentes momentos da abordagem. Por exemplo, um modelo arquitetural pode estar

em estado *inicial* antes de uma determinada atividade, e passar para o estado *refinado* após esta atividade;

- **Papéis**: representam as pessoas que executam ou auxiliam na execução das atividades. Um papel descreve características do indivíduo, e não o indivíduo em si, de forma que uma mesma pessoa pode desempenhar vários papéis, e um papel pode ser desempenhado por várias pessoas. Por exemplo, no papel de especialista do domínio podem estar várias pessoas, cada uma com conhecimentos em pequenas partes do domínio; e

- **Diretrizes**: em alguns casos onde não é possível estabelecer passos explícitos para a execução de uma atividade, a abordagem define diretrizes que podem ajudar ou guiar a sua execução.

Cada atividade é identificada por uma sigla, no formato AD.X, PD.X ou ID.X, onde X é um número sequencial, e AD, PD e ID representam Análise de Domínio, Projeto do Domínio e Implementação do Domínio, respectivamente. Sub-atividades são identificadas incluindo-se outro número sequencial, como por exemplo AD.1.3, que identifica a terceira sub-atividade da primeira atividade da fase de análise de domínio.

Da mesma forma, cada produto de trabalho é identificado por uma sigla, no formato PT.X, onde X é um número sequencial. Caso um produto de trabalho tenha diferentes estados, os mesmos são indicados após a sigla, como por exemplo PT.2.Inicial, que representa o produto de trabalho 2, no estado “Inicial”.

Os produtos de trabalho representam o conhecimento explícito, porém existe também o conhecimento tácito que é utilizado nas atividades da abordagem. Estes são incluídos na entrada das atividades para indicar que aquele conhecimento é necessário para aquela atividade em questão. O conhecimento tácito é representado junto com os produtos de trabalho, porém sem o prefixo PT.

Ao longo dos próximos capítulos, exemplos do domínio de aplicações de autoria de conteúdo para a Web são utilizados para melhor ilustrar as atividades da abordagem. Este domínio representa aplicações Web que permitem a publicação de conteúdo e sua visualização. Nestas aplicações, um autor publica a informação, como notícias e mensagens, que pode ser acessada e visualizada pelos usuários. Portais de notícias, fóruns e *blogs* são alguns exemplos de aplicações deste domínio.

4.4 Modelo de processo

As atividades das fases de análise, projeto e implementação do domínio serão descritas, nos próximos capítulos, de forma sequencial, para facilitar o seu entendimento. Porém, na prática estas atividades acontecem de forma iterativa e interativa, e a sua ordem pode ser alterada para se adequar à realidade de uma organização de software. Nesta seção apresenta-se o modelo de processo de desenvolvimento de software sugerido para a utilização da abordagem.

Um modelo de processo é uma representação abstrata de um processo de software (SOMMERVILLE, 2006). Cada modelo de processo representa um processo sob o ponto de vista de uma perspectiva em particular, e portanto contém apenas informações parciais sobre esse processo. Existem diferentes modelos de processo conhecidos na literatura da Engenharia de Software, como o modelo em cascata, modelo evolutivo, baseado em componentes, incremental, espiral, entre outros (PRESSMAN, 2005). Estes modelos de processo não são mutuamente exclusivos, sendo frequentemente utilizados em conjunto, especialmente no desenvolvimento de grandes sistemas.

A abordagem desta tese é baseada no modelo espiral, proposto originalmente por Boehm (1988). O modelo espiral representa um processo de software em uma espiral, onde todas as atividades são executadas repetidamente a cada iteração. Cada nova volta na espiral acrescenta um desenvolvimento novo, produzindo, por exemplo, novos artefatos de software, ou refinando artefatos já existentes.

A Figura 9 mostra o modelo de processo sugerido para utilização da abordagem para reutilização de software orientada a modelos.

Existem três ciclos básicos neste modelo: o ciclo principal, o ciclo de projeto e o ciclo de implementação. Em cada ciclo, existe uma atividade (AD.1, PD.5 e ID.4, destacadas na Figura 9) que marca um momento de decisão sobre continuar ou não a execução daquele ciclo específico. Cada ciclo é detalhado a seguir.

4.4.1 Ciclo principal do modelo de processo: evolução dos subdomínios

O ciclo principal começa na primeira atividade da análise de domínio (*Atividade AD.1. Planejamento do domínio*) e termina na última atividade da implementação do domínio (*Atividade ID.5. Documentação do domínio*). Cada iteração neste ciclo principal introduz refinamentos em diferentes artefatos, como nos modelos do domínio (modelo de *features*, modelos de casos de uso e de mudança), na arquitetura e seus componentes, e nos artefatos

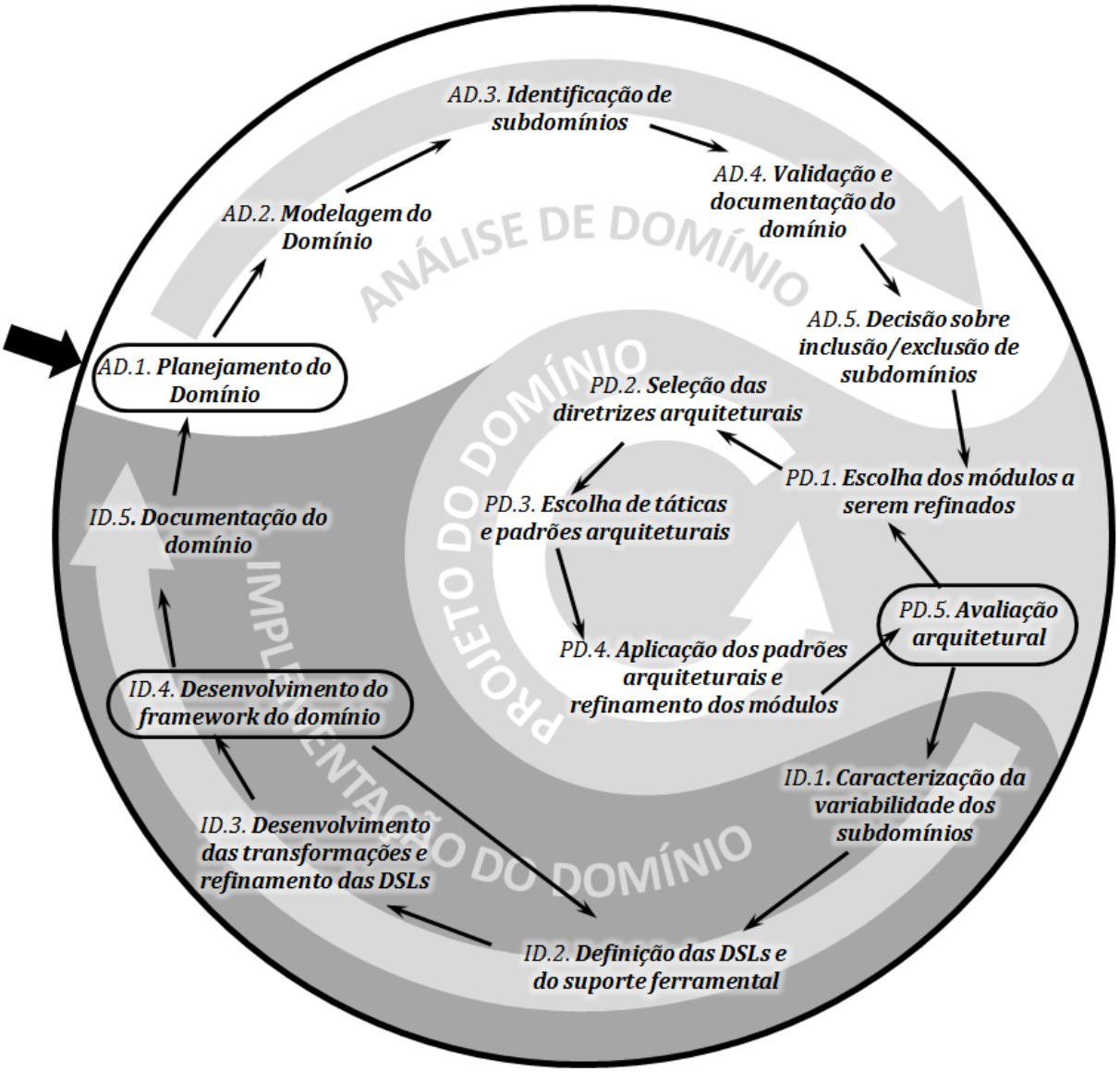

AD.3. Identificação de subdomínios
AD.4. Validação e documentação do domínio
AD.2. Modelagem do Domínio
ANÁLISE DE DOMÍNIO
AD.5. Decisão sobre inclusão/exclusão de subdomínios
AD.1. Planejamento do Domínio
PROJETO DO DOMÍNIO
PD.2. Seleção das diretrizes arquiteturais
PD.1. Escolha dos módulos a serem refinados
PD.3. Escolha de táticas e padrões arquiteturais
ID.5. Documentação do domínio
PD.5. Avaliação arquitetural
PD.4. Aplicação dos padrões arquiteturais e refinamento dos módulos
ID.4. Desenvolvimento do framework do domínio
IMPLEMENTAÇÃO DO DOMÍNIO
ID.1. Caracterização da variabilidade dos subdomínios
ID.3. Desenvolvimento das transformações e refinamento das DSLs
ID.2. Definição das DSLs e do suporte ferramental

Figura 9: Sugestão de modelo de processo para utilização da abordagem

de implementação. Estes refinamentos consistem de melhorias ou correções percebidas durante o desenvolvimento e evolução do domínio.

Mas a atividade mais notável do ciclo principal é a *Atividade AD.5. Decisão sobre inclusão/exclusão de subdomínios*. Conforme será discutido no Capítulo 5, a identificação de subdomínios é pautada por uma incerteza que somente pode ser resolvida através de maiores investigações envolvendo as linguagens, transformações e geradores de código para cada subdomínio. Assim, o ciclo principal gira em torno deste processo investigativo, que culmina com esta atividade, e a decisão nela tomada influencia todas as demais atividades.

O ciclo principal evolui da seguinte forma, a cada iteração;

1. *Atividade AD.1. Planejamento do domínio*: esta atividade consiste da preparação

e planejamento. É necessário determinar se vale a pena investir na construção da infraestrutura reutilizável para o domínio em questão. Para isso, são levantadas e registradas todas as informações referentes ao domínio, são identificadas as *features* do domínio, que servem de base para a elaboração de um plano de desenvolvimento. A cada nova iteração, o plano inicial é revisto, incluindo novas considerações que possam ter sido elicitadas na iteração anterior, durante o projeto e implementação. É também nesta atividade que uma análise de risco é realizada, para determinar se vale a pena continuar investindo no domínio e se novas iterações são necessárias;

2. *Atividade AD.2. Modelagem do domínio*: esta atividade cuida da criação de modelos do domínio, agrupando as *features* identificadas na atividade anterior. Também são desenvolvidos cenários do domínio, com foco na variabilidade. A cada iteração, o analista do domínio refina os modelos de *features* e cenários, considerando-se a eventual inclusão de novos artefatos ou mesmo novos subdomínios;

3. *Atividade AD.3. Identificação de subdomínios*: nesta atividade o objetivo é tentar identificar subdomínios com potencial para automação, com base em diferentes tipos de indícios, como a existência de linguagens, ferramentas e documentação, que resultam em um determinado nível de confiança para cada subdomínio. A cada iteração, o analista do domínio revisita a lista de subdomínios, após a experiência obtida com as implementações e investigações realizadas na iteração anterior. Novos níveis de confiança podem ser atribuídos aos subdomínios identificados, o que pode levar a novas decisões posteriormente. Novos candidatos a subdomínio também podem ser identificados;

4. *Atividade AD.4. Validação e documentação do domínio*: esta atividade consiste na revisão da informação e criação de documentos que descrevem as *features* e modelos construídos nas atividades anteriores. A cada iteração, os documentos existentes são atualizados para refletir eventuais mudanças. Gerência de configuração e controle de versões devem ser utilizados para manter estes documentos organizados;

5. *Atividade AD.5. Decisão sobre inclusão/exclusão de subdomínios*: nesta atividade, o analista do domínio, junto com o especialista e os demais *stakeholders*, analisam os subdomínios identificados com o objetivo de determinar se os mesmos serão incluídos nas fases subsequentes de projeto e implementação. A cada iteração, toda nova informação sobre os candidatos a subdomínio é revista, e novas decisões podem ser tomadas sobre eles;

6. *Fase de projeto de domínio*: na primeira iteração, esta fase cuida da criação de uma arquitetura para o domínio, com base nos requisitos mais relevantes conforme percebidos

pelos *stakeholders* e a existência dos subdomínios. A cada nova iteração do ciclo principal, a fase de projeto produz refinamentos na arquitetura, considerando-se os novos modelos do domínio, novos níveis de decisão sobre os subdomínios ou mais detalhes sobre as diretrizes arquiteturais que possam ter sido percebidos na iteração anterior. Estes refinamentos podem inclusive resultar no surgimento de novos módulos; e

7. *Fase de implementação do domínio*: na primeira iteração, esta fase cuida da implementação das principais partes do domínio, conforme identificado na análise, com a produção de componentes de software, DSLs, transformações e geradores de código. A cada iteração do ciclo principal, são introduzidos refinamentos e mudanças na implementação, para refletir novas decisões de projeto tomadas nesta iteração. Deve-se tomar cuidado especial para que as mudanças não causem inconsistências entre as DSLs, transformações, geradores de código e os componentes.

O resultado de sucessivas iterações no ciclo principal é um crescimento contínuo no número de artefatos do domínio e também no nível de confiança dos subdomínios a serem automatizados. Além disso, cada vez mais subdomínios são descartados, à medida que a experiência demonstra que eles são muito difíceis ou impossíveis de automatizar. Idealmente, no final de várias iterações todos os subdomínios são incluídos ou excluídos do processo, ou seja, a tendência é não existirem candidatos nos níveis intermediários de decisão.

Dependendo do domínio, esta evolução pode apresentar algumas características distintas. Por exemplo, em sistemas críticos, onde é preferível o uso somente de linguagens e ferramentas bem conhecidas, as iterações podem parar após estas terem sido incluídas, sem que haja investigação ou desenvolvimento extra para desenvolver linguagens e ferramentas personalizadas. Em projetos com pouco prazo para entrega, uma única iteração pode ser viável, resultando em vários subdomínios sendo deixados para trás.

Uma sugestão sobre a evolução do domínio, visando reduzir os riscos e problemas associados à adoção do MDD, é incluir, nas primeiras iterações, apenas subdomínios mais conhecidos, com ferramentas e linguagens de modelagem disponíveis. Desta forma, pode-se perceber os benefícios do MDD sem a necessidade de um investimento inicial maior, além de proporcionar uma evolução mais rápida nestas primeiras iterações.

4.4.2 Ciclo de projeto do modelo de processo: evolução arquitetural

Dentro do ciclo principal, a fase de projeto também ocorre de forma iterativa. A cada iteração no ciclo de projeto, novos módulos são identificados, formando a arquitetura do

domínio. Trata-se de um processo de refinamento dirigido por padrões. A divisão em módulos ocorre de acordo com a instanciação de um ou mais padrões que atendem às diretrizes arquiteturais.

O projeto se inicia com uma divisão principal, envolvendo todo o domínio. Esta divisão produz os módulos principais do sistema. Por exemplo, se for adotado o padrão em camadas (BUSCHMANN et al., 1996), a primeira divisão resulta na identificação das camadas do domínio. Se for adotado o padrão MVC (*Model-View-Controller*) (BUSCHMANN et al., 1996), a primeira divisão resulta na identificação dos elementos de modelo, de visão e de controle, e assim por diante.

As iterações seguintes refinam cada módulo de acordo com o padrão escolhido. Por exemplo, ao se aplicar o padrão *Observer* (GAMMA et al., 1995) em um módulo, serão identificados novos módulos (ou classes), como o *Sujeito*, *Sujeito Concreto*, *Observador* e *Observador Concreto*.

No caso desta abordagem focada em reutilização, os padrões visam, além de identificar novos módulos, adicionar o suporte à variabilidade prevista durante a análise. Além disso, tendo foco também no MDD, cada refinamento identifica não somente módulos de software convencional, como classes e métodos, mas também elementos como transformações e geradores de código, além de DSLs e ferramentas de modelagem. Todos estes também fazem parte da arquitetura.

Assim, a evolução do ciclo de projeto ocorre da seguinte maneira, para cada atividade:

1. *Atividade PD.1. Escolha dos módulos a serem refinados*: a cada iteração, esta atividade escolhe novos módulos para serem refinados. Obviamente, na primeira iteração ainda não existem módulos, e a escolha é o domínio todo. As demais atividades são executadas individualmente para cada módulo aqui escolhido;

2. *Atividade PD.2. Seleção das diretrizes arquiteturais*: esta atividade identifica, para cada módulo sendo refinado nesta iteração, os principais requisitos de software que podem influenciar a arquitetura naquele exato local referente ao módulo. Para cada módulo seleciona-se, dentre todos os requisitos e objetivos de negócio do domínio, aqueles que dizem respeito somente ao módulo em questão, e que possuem impacto em sua arquitetura. Estas são as chamadas diretrizes arquiteturais;

3. *Atividade PD.3. Escolha de táticas e padrões arquiteturais*: nesta atividade o projetista do domínio escolhe, para cada módulo sendo refinado, e com base nas diretrizes

selecionadas para aquele módulo, táticas e padrões arquiteturais a serem aplicados no refinamento do módulo;

4. *Atividade PD.4. Aplicação dos padrões arquiteturais e refinamento dos módulos*: esta é a atividade onde é efetivamente realizado o refinamento dos módulos. A cada iteração, com a aplicação das táticas e padrões escolhidos, os módulos são refinados, possivelmente resultando na identificação de novos módulos e relacionamentos; e

5. *Atividade PD.5. Avaliação arquitetural*: esta atividade busca avaliar a arquitetura projetada frente aos seus requisitos. A cada iteração, novas arquiteturas produzidas são avaliadas, e o resultado da avaliação serve de entrada para uma nova iteração no ciclo de projeto.

O resultado das sucessivas iterações no ciclo de projeto é a produção de uma arquitetura para o domínio, capaz de dar suporte às variabilidades identificadas durante a análise. Dependendo do número de módulos identificados, são necessárias mais ou menos iterações. A tendência é que o número de iterações no ciclo de projeto diminua a cada iteração do ciclo principal, uma vez que restam cada vez menos módulos para serem refinados.

4.4.3 Ciclo de implementação do modelo de processo: produção dos artefatos reutilizáveis e documentação

A fase de implementação também possui iteratividade, que ocorre em um ciclo menor, composto pelas atividades ID.2, ID.3 e ID.4. As iterações desta fase são intrínsecas a qualquer atividade de codificação, exigindo re-trabalho à medida que a implementação avança.

No caso desta abordagem, a iteratividade visa coletar informações durante a codificação, realimentando a atividade de definição das linguagens específicas de domínio. Isto porque no contexto do MDD, uma DSL não é somente um mecanismo de comunicação de idéias. Ela deve ser não-ambígua e rica o suficiente para servir de entrada para transformações e geradores de código. Para isso, são necessários detalhes e ajustes que só são percebidos após a experiência de codificação.

Além disso, a implementação envolve diferentes subdomínios. Cada iteração cuida do desenvolvimento dos artefatos de implementação para um único subdomínio. Assim, caso existam dez subdomínios, por exemplo, serão necessárias dez iterações neste ciclo.

A fase de implementação evolui da seguinte maneira:

1. *Atividade ID.1. Caracterização da variabilidade dos subdomínios*: esta atividade, que está fora do ciclo iterativo da fase de implementação, identifica o tipo de variabilidade inerente a cada subdomínio. É executada uma única vez a cada iteração do ciclo principal, e seu objetivo é definir o tipo de suporte em termos de DSL e ferramentas que será necessário para cada subdomínio. É importante ressaltar que nem todos os subdomínios identificados na análise irão passar pela fase de implementação. Somente são considerados os subdomínios selecionados para investigação ou produção durante esta iteração do ciclo principal, conforme decidido na atividade AD.5 da fase de análise;

2. *Atividade ID.2. Definição das DSLs e do suporte ferramental*: esta atividade cuida do desenvolvimento de uma DSL para um subdomínio, assim como o seu suporte ferramental (ferramentas de modelagem e editores). A cada iteração, mais subdomínios são implementados, até que todos aqueles selecionados na atividade AD.5 estejam implementados ou devidamente investigados;

3. *Atividade ID.3. Desenvolvimento das transformações e refinamento das DSLs*: nesta atividade, são desenvolvidas transformações de software e geradores de código para os subdomínios, e as DSLs definidas na atividade ID.2 são refinadas a partir da experiência com a implementação. A cada iteração, novas transformações e geradores são desenvolvidos, até que todos os subdomínios selecionados na atividade AD.5 tenham transformações e geradores, ou foram devidamente investigados. Este desenvolvimento é baseado em uma implementação de referência, que serve como exemplo para a construção dos geradores de código;

4. *Atividade ID.4. Desenvolvimento do framework do domínio*: esta atividade cuida do refinamento da implementação de referência produzida na atividade ID.3, produzindo um *framework* do domínio. A cada iteração, o *framework* evolui com o suporte à variabilidade de mais subdomínios; e

5. *Atividade ID.5. Documentação do domínio*: esta atividade, que a exemplo da atividade ID.1 está fora do ciclo iterativo da fase de implementação, cuida da documentação dos artefatos desenvolvidos nesta iteração do ciclo principal.

Após a fase de implementação, tem-se como resultado um conjunto de artefatos de implementação que atendem ao projeto realizado na fase anterior. Estes artefatos incluem DSLs, transformações, geradores de código e um *framework* do domínio, e seguem a arquitetura definida no projeto.

É importante ressaltar que nem sempre as iterações dos ciclos da fase de implementação produzem artefatos que farão parte do domínio. Como o suporte automatizado a subdomínios é incerto e exige investigação, o resultado da implementação pode ser insatisfatório, de modo que alguns subdomínios podem não ser incluídos no processo. Esta decisão irá ocorrer na próxima iteração do ciclo principal, no final da fase de análise, subsidiada pelas experiências obtidas com a fase de implementação.

4.5 Abrangência da abordagem

A abordagem definida nesta tese não cobre todo o ciclo de vida do software, pois são incluídas somente práticas de engenharia, que dizem respeito à utilização do MDD como meio de aumentar a reutilização de software, e algumas práticas referentes ao gerenciamento. A Figura 10 ilustra a abrangência da abordagem, em relação aos modelos de maturidade em reutilização e MDD apresentados nas Seções 2.1.2 e 2.2.3.

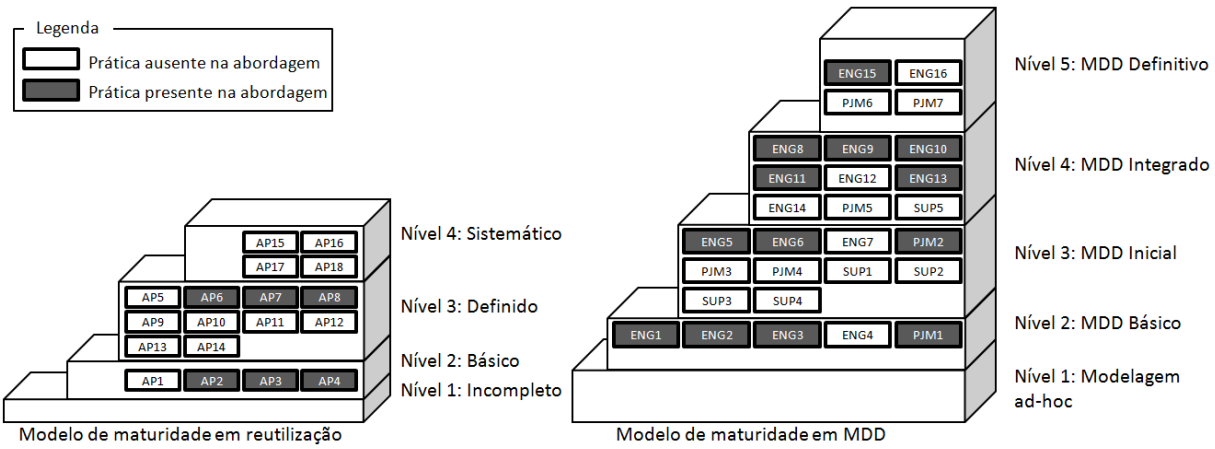


Figura 10: Abrangência da abordagem, em relação aos modelos de maturidade em reutilização e MDD

No total, as atividades da abordagem aderem a 6 práticas do modelo de maturidade em reutilização e 13 práticas do modelo de maturidade em MDD, conforme descrito a seguir[1]:

- **Reutilização**
  - **AP2 - Planejamento de reutilização**: uma das atividades iniciais da abordagem consiste no planejamento da reutilização, incluindo possíveis riscos, e a adoção desta abordagem pode ser considerada como uma decisão de planejamento, assim como a definição de seu modelo do ciclo de vida;

[1] Uma discussão mais detalhada sobre cada prática dos modelos de maturidade e a sua relação com a abordagem aqui proposta é apresentada no Apêndice B.

- **AP3 - Definição dos objetivos da reutilização**: a abordagem contém uma atividade para definição dos objetivos da reutilização;
- **AP4 - Implementação do domínio**: uma das fases da abordagem, que visa implementar um domínio utilizando técnicas do MDD;
- **AP6 - Integração com o ciclo de vida do software**: esta prática consiste justamente na definição desta abordagem e seus elementos;
- **AP7 - Análise de domínio**: a abordagem realiza a análise de domínio voltada para o MDD;
- **AP8 - Projeto de domínio**: a abordagem engloba as práticas relacionadas ao projeto de domínio, incluindo projeto arquitetural e suporte à variabilidade;

- **MDD**

  - **ENG1 - Decidir sobre convenções de modelagem**: a abordagem possui uma atividade na qual são analisadas as possibilidades de linguagens a serem utilizadas na modelagem;
  - **PJM1 - Decidir sobre ferramentas de modelagem**: assim como na atividade ENG1, a abordagem também busca analisar ferramentas existentes para serem utilizadas;
  - **ENG2 - Desenvolver modelo técnico**: a abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver modelos técnicos;
  - **ENG3 - Gerar código a partir do modelo técnico**: a geração de código é um dos principais itens da abordagem, e portanto esta prática está plenamente implementada;
  - **ENG5 - Desenvolver modelo independente de plataforma**: a abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver modelos independentes de plataforma;
  - **ENG6 - Desenvolver transformações técnicas modelo-para-texto**: a abordagem tem atividades para transformações modelo-para-texto, visando dar suporte à variabilidade do domínio;
  - **PJM2 - Modelar o processo de MDD do projeto**: a abordagem está completamente modelada, e portanto pode ser utilizada como modelo do processo;

- **ENG8 - Desenvolver metamodelo centrado na arquitetura**: o projeto arquitetural voltado ao MDD está presente na abordagem, que tem atividades para que o metamodelo seja centrado na arquitetura;
- **ENG9 - Desenvolver metamodelo independente de plataforma**: a abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver metamodelos independentes de plataforma;
- **ENG10 - Desenvolver modelo de negócios**: a abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver modelos de negócio;
- **ENG11 - Desenvolver transformações modelo-para-modelo**: as transformações modelo-para-modelo estão previstas na abordagem, como uma forma de organização e estruturação dos geradores de código;
- **ENG13 - Integrar desenvolvimento de produto com desenvolvimento de infraestrutura para uma família de produtos**: esta é a preocupação central da abordagem;
- **ENG15 - Desenvolver linguagens específicas de domínio**: a abordagem possui uma atividade específica para o desenvolvimento de DSLs no contexto da reutilização;

Conforme pode ser constatado, em termos de reutilização a abordagem cobre principalmente as práticas de engenharia, incluindo análise, projeto e implementação do domínio, além de algumas práticas de gerenciamento isoladas, como planejamento e definição de objetivos. Considerando-se este modelo, a presente abordagem não garante que uma organização atinja um nível maior do que o nível 1 de maturidade.

Em termos de MDD, a abordagem implementa quase todas as práticas de engenharia, com exceção de quatro práticas. Como na reutilização, poucas práticas de gerenciamento e suporte do MDD foram incluídas. Considerando-se este modelo, a abordagem não garante que uma organização atinja um nível maior do que o nível 1 de maturidade. Porém, caso a organização defina uma atividade para geração automática de documentação (prática ENG4), ela passa ao nível 2, pois esta é a única prática do nível 2 não implementada pela abordagem.

A comparação com estes modelos de maturidade visa somente deixar clara a abrangência da abordagem com relação ao que existe, em termos de processo de software, conforme identificado pelos autores destes modelos. Estes modelos não foram utilizados como base para a definição da abordagem, mesmo porque apesar de representarem iniciativas promissoras, lideradas por importantes instituições de todo o mundo, ainda não são consolidados.

A decisão sobre quais práticas atender e quais não atender foi portanto baseada na literatura e nos estudos de caso, conforme descrito na Seção 4.2. Desta maneira, pode-se obter uma cobertura mais completa do ciclo de vida, desde a análise até a implementação, ainda que isto signifique que nenhum dos primeiros níveis de maturidade destes modelos seja alcançado.

4.6 Considerações finais

Neste capítulo apresentou-se uma visão geral da abordagem, seus objetivos, sua estrutura e seu escopo. Em particular, nota-se que a abordagem está mais focada nas práticas relacionadas à engenharia, deixando de lado a maioria das atividades de suporte e planejamento, que são importantes, mas que estão fora do escopo desta tese.

Foi também apresentado um possível modelo de processo para a utilização da abordagem de reutilização orientada a modelos. Esta é apenas uma sugestão, que busca combinar as atividades de forma natural. O foco deste modelo é a iteratividade, forçada pelo aspecto investigativo que deriva da incerteza sobre as possibilidades de automação dos subdomínios.

Porém, este modelo pode ser adaptado para refletir outras necessidades da organização, de forma a reforçar aspectos que aqui foram ignorados. Por exemplo, como esta tese tem foco mais técnico, foram omitidos mais detalhes sobre gerenciamento de riscos, atividades de gerenciamento e controle, melhoria do processo, uso de métricas, entre outros pontos igualmente importantes em um projeto de software.

A seguir as três fases da abordagem são apresentadas em maiores detalhes, em capítulos separados, e com cada atividade sendo descrita de modo sequencial, visando facilitar seu entendimento.

5 *Análise de domínio orientada a modelos*

A análise de domínio envolve a identificação dos principais conceitos e elementos de um domínio e a determinação de seu escopo, isto é, o que será incluído e o que será excluído do conjunto dos artefatos a serem desenvolvidos para o domínio. Além disso, visa identificar as comunalidades e variabilidades, ou seja, os pontos que são comuns a todas as aplicações do domínio e os pontos que variam. Como discutido na Seção 3.1.1, uma das abordagens mais comuns para esta tarefa é a modelagem de *features* (KANG et al., 1990; KANG; LEE; DONOHOE, 2002).

Um dos principais desafios em projetar e implementar um domínio descrito em termos de suas *features* está em como mapear estas *features* para a arquitetura e o código, considerando todos os relacionamentos e restrições envolvendo as mesmas.

A existência de uma *feature* pode levar a diferentes tipos de impacto no produto final. Em casos mais simples, é possível "ligar" ou "desligar" uma *feature* através de um parâmetro em um componente, de forma que atribuindo um valor para este parâmetro, a *feature* estará presente ou ausente. Por exemplo, considere um domínio de automóveis, onde existem três tipos de motores: Diesel, Gasolina e Álcool.

Em um sistema mais simples, como por exemplo um sistema de uma companhia de aluguel de automóveis, o tipo do motor pode ser simplesmente representado com um atributo do tipo inteiro, onde 0=Diesel, 1=Gasolina e 2=Álcool. Escolher um valor apropriado para este atributo é o suficiente para que a *feature* seja incluída na aplicação.

Em um sistema mais complexo, porém, como por exemplo um sistema de uma companhia de transportes, a existência de motores a Diesel, mais caros, pode requerer a presença de um dispositivo GPS para aumentar a segurança. Outro exemplo seria o dispositivo de freios com atuação eletrônica precisar ser calibrado de uma maneira específica, para funcionar adequadamente com um motor a Diesel. A implementação destas restrições não é tão imediata quanto no exemplo do sistema de aluguel de automóveis, e normalmente não

pode ser encapsulada dentro de um único componente, necessitando ser realizada durante o desenvolvimento da aplicação.

É neste ponto que as técnicas do MDD podem ser utilizadas. Ao invés de deixar que estas tarefas sejam executadas pelo desenvolvedor da aplicação, o conhecimento sobre como implementar estas restrições é encapsulado em transformações MDD e linguagens específicas de domínio (LEDECZI et al., 2001). Em um cenário ideal, tudo o que um desenvolvedor precisa fazer é escolher quais *features* estarão presentes, especificar alguns parâmetros, e uma implementação é automaticamente gerada.

Obviamente, este cenário ainda está longe da realidade, já que nem todo código pode ser completamente gerado. Porém, normalmente há partes do domínio onde isto não somente é possível, mas pode levar a enormes ganhos de produtividade, aumentando o nível de reutilização. Porém, como identificar que partes do domínio podem ser automatizadas?

O argumento desta tese é que esta atividade deve ser parte da análise de domínio. Enquanto analisando as *features* do domínio, devem existir maneiras para identificar o potencial para automação em alguns subdomínios, numa preparação para o desenvolvimento de linguagens e transformações, que irá acontecer nas fases seguintes (LUCRÉDIO et al., 2008).

5.1 Objetivos da análise de domínio

A fase de análise de domínio faz parte da engenharia de domínio, e é responsável por coletar informações do domínio para as fases seguintes de projeto e implementação. Nesta fase, os seguintes objetivos são almejados:

- **Coletar, registrar e documentar todas as informações disponíveis sobre o domínio**: estas informações são coletadas de diversas fontes, incluindo o conhecimento de especialistas e desenvolvedores experientes com o domínio, documentações e padrões sobre o domínio, e também aplicações pertencentes ao domínio. Neste último caso, utiliza-se toda informação disponível, desde os próprios executáveis até manuais e código-fonte, caso possível;

- **Definir o escopo do domínio a ser desenvolvido**: um domínio pode alcançar uma vasta gama de aplicações, cada uma com foco específico em determinada parte. Além disso, a própria definição do que está dentro ou fora de um domínio pode variar de acordo com a visão dos especialistas, desenvolvedores, ou demais envolvidos, pois cada um destes *stakeholders* possui seus próprios interesses distintos. Dessa forma, é necessário definir

com exatidão o escopo a ser desenvolvido. Para tal definição, ao invés de buscar atender a artefatos genéricos do domínio, de acordo com a percepção/intuição de especialistas, esta abordagem foca em um conjunto finito de aplicações ou produtos que sejam de interesse da organização (CLEMENTS; NORTHROP, 2002). Dessa forma, são menores as chances de um artefato poder ser reutilizado em aplicações não previstas, mas aumenta-se consideravelmente o seu nível de reutilização dentro deste escopo;

- **Criar modelos do domínio**: modelos expressam os conceitos do domínio de forma a facilitar o seu entendimento por parte dos projetistas e desenvolvedores. Além disso, os modelos devem explicitar as variabilidades e comunalidades dentro do domínio (KANG et al., 1990; JACOBSON; GRISS; JONSSON, 1997; GRISS; FAVARO; D'ALESSANDRO, 1998; KANG et al., 1998; KANG; LEE; DONOHOE, 2002), de forma a facilitar o projeto de software reutilizável; e

- **Identificar os subdomínios onde a automação é possível**: nesta abordagem, o desenvolvimento orientado a modelos é utilizado para aumentar o nível de reutilização, principalmente na implementação das variabilidades. Ao invés de deixar esta tarefa para ser realizada pelos desenvolvedores de aplicações, o conhecimento de como implementar as variabilidades é encapsulado em ferramentas de modelagem e transformações automáticas de software (LEDECZI et al., 2001). O objetivo final é que, após a engenharia de domínio, um desenvolvedor possa simplesmente configurar o produto desejado e acionar as transformações que automaticamente geram uma implementação válida. Assim, esta fase de análise também tem como objetivo identificar subdomínios onde a automação é possível.

A abordagem aqui proposta é uma evolução do método descrito por Almeida et al. (2006). A diferença é que aqui existe a preocupação com a identificação dos subdomínios onde a automação é possível, e portanto a análise e tratamento das variabilidades é realizada de forma diferenciada, com este objetivo em mente.

5.2 Atividades da análise de domínio

A análise de domínio começa com uma atividade de planejamento (Atividade AD.1) que visa coletar informações para decidir se vale a pena investir no domínio e definir o escopo do domínio a ser construído. Em seguida, a modelagem do domínio (Atividade AD.2) cuida da criação de modelos do domínio, que descrevem as funções, as comunalidades e variabilidades do mesmo. A seguir, é feita a identificação de possíveis subdomínios (Atividade AD.3),

visando determinar os locais onde é possível a automação através das técnicas do MDD. A próxima atividade cuida da validação e documentação do domínio (Atividade AD.4), reunindo as informações disponíveis até o momento. Por fim, ocorre uma decisão sobre quais possíveis subdomínios identificados devem ser incluídos ou excluídos do processo, e quais precisam ser investigados de maneira mais aprofundada (Atividade AD.5).

A seguir estas cinco atividades são descritas em maiores detalhes.

Atividade AD.1. Planejamento do domínio

Papéis: analista do domínio, especialista do domínio, especialista de mercado

Entradas: *Informações sobre sistemas do domínio*, *Conhecimento do especialista*, *Informações sobre stakeholders*

Saídas: *PT.1.Inicial. Planejamento do domínio*, *PT.2.Inicial. Mapa de aplicações*.

Descrição: a primeira etapa cuida da preparação e planejamento. É necessário determinar se vale a pena investir na construção da infraestrutura reutilizável para o domínio em questão. Para isso, nesta atividade são levantadas e registradas todas as informações referentes ao domínio.

Inicialmente, é realizada uma tarefa de *preparação (Sub-atividade AD.1.1)*, onde são realizadas diversas análises, tais como identificação dos *stakeholders*, análise do mercado, coleta de informações, e definição de objetivos e restrições, de acordo com as análises e informações coletadas. Em seguida, é realizada a *definição do escopo (Sub-atividade AD.1.2*), onde as aplicações existentes, futuras e potenciais são identificadas, e o escopo do domínio é definido.

Sub-atividade AD.1.1. Preparação

O objetivo da preparação é levantar e registrar todas as informações necessárias para o planejamento. Para isso, o analista do domínio realiza as seguintes tarefas.

Análise dos stakeholders: compreende a identificação dos diferentes *stakeholders*, seus papéis e interesses dentro do processo.

Definição dos objetivos: corresponde à definição dos objetivos, de acordo com os interesses dos *stakeholders* para o projeto.

Definição de restrições: compreende a definição das restrições impostas pela organização

e pelas condições de mercado, deixando claro o que está além do escopo do processo.

Análise de mercado (se aplicável): esta tarefa não-trivial, que pode ser realizada ou não, dependendo dos custos, complexidade e maturidade da organização com relação ao domínio, consiste na pesquisa e análise sistemáticas dos fatores que determinam o sucesso do domínio no mercado. Junto com um especialista de mercado, o analista do domínio coleta informações sobre o negócio, estudos e análises da competição, segmentação de mercado, planos de negócios, e a integração desta informação em um plano estratégico de negócios coeso (CLEMENTS; NORTHROP, 2002).

Coleta de dados: é a tarefa realizada pelo analista do domínio, junto com o especialista do domínio, responsável por elicitar e examinar todo tipo de conhecimento sobre o domínio em foco. Inclui a análise de toda a documentação existente (planos de projetos, manuais de usuário, modelos, dicionários de dados), aplicações existentes e o conhecimento do especialista do domínio.

Toda a informação deve ser organizada de forma a facilitar sua posterior consulta. Não existe uma estrutura fixa para esta tarefa, que depende de condições específicas do projeto. Enquanto organizações que possuam sistemas de repositório possam utilizar um método automático de armazenamento e busca, simples pastas e arquivos podem também ser utilizados. De qualquer forma, as seguintes diretrizes podem ser úteis:

- Definir e manter uma estrutura não muito detalhada dos dados coletados, mas que seja consistente. Deve ser possível ao menos descartar grandes quantidades de informação durante uma busca;
- Sempre que possível, incluir referências para as fontes originais; e
- Manter uma lista sobre as pessoas envolvidas e consultadas durante o processo, com suas informações de contato, para eventuais consultas.

Sub-atividade AD.1.2. Definição do escopo

O objetivo desta sub-atividade é identificar as *features* que estarão presentes na futura arquitetura do domínio.

Inicialmente, o analista do domínio, com base nas informações sobre sistemas do domínio e *stakeholders*, identifica as aplicações a serem contempladas no domínio. Existem três tipos de aplicações distintas: *aplicações existentes* (aplicações que foram desenvolvidas antes do início do processo de análise de domínio), *aplicações futuras* (aplicações para as quais os requisitos

estão bem claros, porém o desenvolvimento ainda não foi iniciado) e *aplicações potenciais* (aplicações para as quais ainda não existem requisitos definidos, mas que são vistas como relevantes).

Após a identificação das aplicações, a lista de *features* é desenvolvida. A identificação de *features* envolve abstrair o conhecimento obtido com os especialistas do domínio e outros documentos, tais como livros, manuais de usuário, documentos de projeto e código-fonte (LEE; KANG; LEE, 2002). Porém, o volume de informação de um domínio tende a ser muito grande. Neste sentido, quatro diretrizes (ALMEIDA et al., 2006) podem ser úteis:

D_1. *Analisar as terminologias usadas no domínio para identificar features*: em alguns domínios mais maduros, especialistas normalmente utilizam terminologia padronizada para comunicar suas idéias, necessidades e problemas. Assim, utilizando-se os mesmos termos padrões na identificação de *features*, pode-se acelerar a comunicação entre o analista do domínio e os provedores de informação (especialista do domínio e usuários finais).

D_2. *Categorizar as features*: as *features* devem ser categorizadas, utilizando, por exemplo as quatro categorias descritas na Seção 3.1.1: *features* de capacitação, *features* de tecnologia do domínio, *features* de técnicas de implementação e *features* do ambiente operacional. Para organizações com pouca experiência na análise de domínio, ou que estão trabalhando com domínios instáveis e pouco amadurecidos, é aconselhável concentrar-se nas *features* de capacitação. Após certa maturidade com o domínio ser alcançada, pode-se passar para as demais *features*.

D_3. *Tentar primeiro encontrar diferenças entre as aplicações de um domínio, para só então tentar identificar as partes em comum*: aplicações de um mesmo domínio compartilham um alto grau de funções em comum (COPLIEN; HOFFMAN; WEISS, 1998). Portanto, o espaço em comum deve ser consideravelmente maior do que as diferenças, e por consequência é mais fácil encontrar as diferenças primeiro. A estratégia é, inicialmente, identificar as aplicações existentes, e listar as *features* que caracterizam cada uma. Encontradas as diferenças, é mais fácil identificar as *features* comuns.

D_4. *Não identificar todos os detalhes de implementação que não se distinguem entre as aplicações do domínio*: os desenvolvedores tendem a listar todos os detalhes de implementação e identificá-los como *features*, mesmo que não haja variações entre eles. Mas é importante notar que um modelo de *features* não é um modelo de requisitos, que expressa os detalhes de funções internas. Por essa razão, esta atividade deve ser realizada pelo analista do domínio, que tende a abstrair detalhes de implementação.

Por fim, as aplicações e suas *features* são organizadas na forma de um mapa de

aplicações. Neste mapa, colunas e linhas são utilizadas para representar *features* e aplicações, respectivamente. Algumas vezes, é comum dividir uma *feature* em um conjunto de *sub-features*, com um relacionamento entre eles, para facilitar seu entendimento.

Com base neste mapa de aplicações, é realizada uma análise para identificar quais as *features* estarão presentes no domínio, e quais não estarão. Este é um processo iterativo, e não precisa ser definido em definitivo logo na primeira iteração, sendo refinado sucessivamente (GRISS; FAVARO; D'ALESSANDRO, 1998).

Os Quadros 1 e 2 mostram um exemplo de como elaborar o mapa de aplicações. No Quadro 1 são mostradas duas aplicações do domínio de autoria de conteúdo para Web e suas *features*. Nota-se que neste nível as *features* são identificadas de modo bastante informal, pois o objetivo é fazer o levantamento inicial. Após o levantamento ter sido feito com todas as aplicações definidas para o escopo (existentes, futuras e potenciais), é feita uma compilação das *features*, de forma a identificar aquelas mais ou menos comuns do domínio. Nesta compilação, algumas *features* podem ter seus nomes alterados, enquanto outras podem ser generalizadas ou suprimidas, para representar de forma mais correta os conceitos do domínio. O Quadro 2 mostra um exemplo de compilação envolvendo quatro aplicações deste domínio.

Aplicação: http://www.americanas.com.br				
Descrição: Portal web de e-commerce, para compra de produtos pela Internet				
Features:				
Menu	Departamentos			
	Categorias			
	Produtos	Página	Texto	
			Imagens	
			Links	
Busca	Todo o site			
	Departamentos			
Carrinho de compra	Meus pedidos			
	Cadastro			
Administração	Cadastro de produtos			
	Cadastro de categorias			

Aplicação: http://forum.clubedohardware.com.br				
Descrição: Fórum para troca de experiências com hardware				
Features:				
Lista de fóruns	Fóruns	Tópicos	Mensagens	Texto
				Imagens
				Links
Cadastro				
Busca	Simples			
	Avançada			
Administração	Moderação			
	Cadastro de fóruns			

Quadro 1: Identificação inicial das *features* de duas aplicações do domínio de autoria de conteúdo para Web

Aplicações	Features					
	Página	Busca		Carrinho	Administração	
		Total	Parcial		Cadastros	Moderação
http://www.submarino.com.br	X	X	X	X	X	
http://www.americanas.com.br	X	X	X	X	X	
http://forum.clubedohardware.com.br	X	X	X		X	X
http://smeira.blog.terra.com.br/	X	X				X

Quadro 2: Compilação das *features* de quatro aplicações do domínio de autoria de conteúdo para Web

O mapa de aplicações e a lista de *features*, que é extraída diretamente do mapa de aplicações, servem de guia para a próxima atividade, de modelagem do domínio.

Atividade AD.2. Modelagem do domínio

Papéis: analista do domínio, especialista do domínio

Entradas: *Informações sobre sistemas do domínio*, *Conhecimento do especialista*, *Informações sobre stakeholders*, *PT.1.Inicial. Planejamento do domínio*, *PT.2.Inicial. Mapa de aplicações*.

Saídas: *PT.3.Inicial. Modelagem do domínio*.

Descrição: esta atividade cuida da criação de modelos do domínio, agrupando as *features* identificadas na atividade anterior. Também são desenvolvidos cenários do domínio, com foco na variabilidade.

Esta atividade é responsável por criar modelos expressivos do domínio, através da descrição de suas *features* e seus relacionamentos, com foco nas variabilidades do domínio. Enquanto a atividade anterior se preocupou com o *escopo*, aqui a preocupação é com a estrutura interna do domínio.

Nesta abordagem, a técnica de modelagem de *features* é utilizada (Seção 3.1.1). O analista do domínio agrupa as *features* identificadas na atividade anterior em uma hierarquia gráfica que captura relacionamentos estruturais lógicos entre as *features*.

Existe na literatura um conjunto de diretrizes que podem auxiliar nesta atividade (ALMEIDA et al., 2006; LEE; KANG; LEE, 2002). No contexto do desenvolvimento orientado a modelos, a modelagem das *features* e seus relacionamentos é de grande importância, pois ela indicam pontos de especial interesse para os transformadores e ferramentas de modelagem, pois especificam os procedimentos que mapeiam as abstrações em nível de domínio.

A Figura 11 mostra o modelo de *features* criado para o domínio de aplicações de autoria de conteúdo para a Web. Nota-se que as *features* identificadas na atividade anterior foram incluídas, além dos relacionamentos entre elas e novas *features*, como *link*, *Formulário* e *Relatório*.

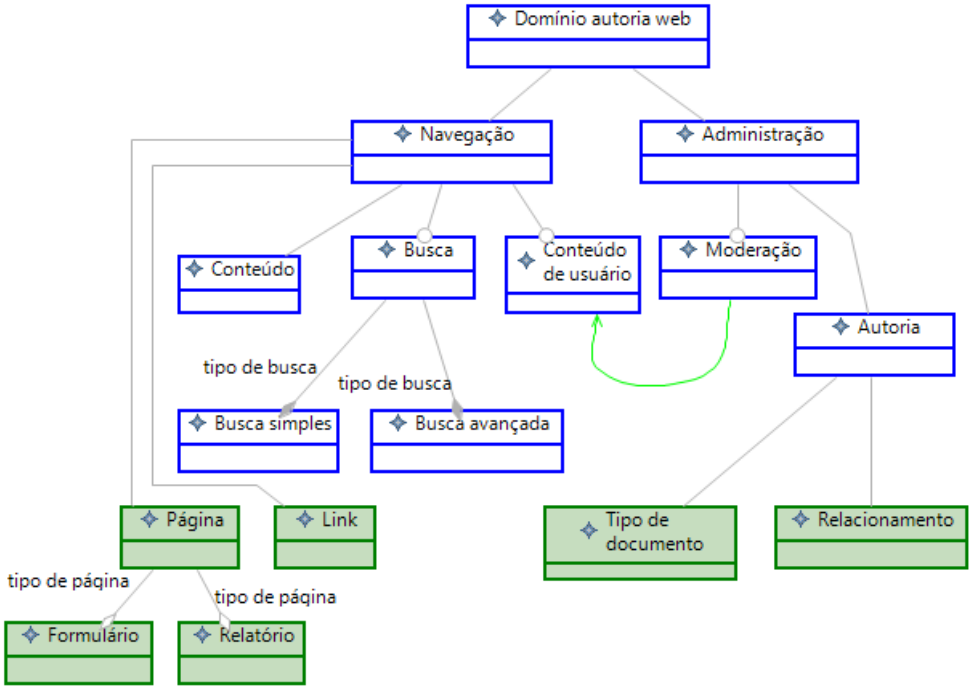


Figura 11: Modelo de *features* do domínio web de autoria de conteúdo, segundo a notação da ferramenta ToolDay (LISBOA, 2007)

Sub-atividade AD.2.1. Mapeamento da variabilidade em cenários

O objetivo desta sub-atividade é mapear os pontos de variação, identificados na modelagem de *features*, para cenários do domínio.

A modelagem de *features* identifica os pontos de variabilidade em termos estáticos. Porém, o impacto destas variabilidades nas funções (comportamento) realizadas pelas aplicações do domínio não fica explícito nos diagramas de *features*. Assim, esta atividade cuida do mapeamento entre a variabilidade no nível de *features* e os cenários que descrevem a funcionalidade, de forma que é possível determinar o que muda no comportamento do domínio.

Para tanto, utiliza-se o conceito de casos de mudança (ECKLUND JR; DELCAMBRE; FREILING, 1996), uma técnica similar aos casos de uso e que busca prever possíveis mudanças de requisitos durante a análise. Aqui, é utilizada para representar a variabilidade em termos de possíveis cenários, cobrindo tanto requisitos funcionais como não-funcionais. Essencialmente, é um processo similar ao seguido pelo método PuLSE (*ProdUct-Line Software Engineering*)

(DEBAUD; FLEGE; KNAUBER, 1998), com a diferença de que aqui são providos mais detalhes sobre sua execução.

Um caso de mudança é um cenário que descreve um ponto de variação no domínio, com foco nas mudanças trazidas pela presença de uma ou mais variantes. Para cada caso de mudança, são registradas as *features* relacionadas e os cenários (casos de uso ou mudança) que são afetados. Por exemplo, considere o modelo de *features* da Figura 11, e o caso de uso referente à *feature* de navegação, ilustrado no Quadro 3.

Nome: **UC002. Navegação**
Atores: **Usuário final**
Fluxo normal: 1. Usuário acessa página da aplicação 2. Aplicação exibe conteúdo da página, caso necessário realiza consulta nos cadastros para exibir o conteúdo 3. Usuário clica em links para navegar para outra página
Fluxos alternativos: Não há
Features relacionadas: **Navegação, Conteúdo, Página, Link, Conteúdo de Usuário**

Quadro 3: Exemplo de caso de uso do domínio web

Conforme mostra a Figura 11, o domínio contempla uma *feature* opcional de busca (círculo branco no final do conector). Um caso de mudança que descreve essa *feature* opcional é apresentado no Quadro 4.

Nome: **CM001. Realizar busca simples**
Atores: **Usuário final**
Fluxo normal: 1. Usuário informa "string" de busca 2. Aplicação realiza busca sobre todo o conteúdo cadastrado 3. Aplicação exibe lista com o conteúdo que corresponde à "string" informada 4. Usuário clica sobre um resultado e continua navegação
Fluxos alternativos: 3. Não existe conteúdo associado à "string" informada 3.1. Aplicação exibe mensagem informando que não foi encontrado nenhum conteúdo
Features relacionadas: **Navegação, Busca, Busca simples**
Cenários afetados: **UC002. Navegação**

Quadro 4: Exemplo de caso de mudança para a *feature* opcional de busca

Neste exemplo, o caso de mudança **CM001. Realizar busca simples** descreve um cenário de uso considerando-se a presença desta *feature* opcional. São também indicadas as *features* relacionadas, e os casos de uso afetados.

Mesmo um caso de mudança pode sofrer o impacto de algum outro ponto de variação. Neste exemplo, a busca pode ser simples ou avançada (*features* alternativas, indicadas pelos losangos nos conectores, no diagrama da Figura 11). O caso de mudança **CM001. Realizar busca simples** descreve o cenário correspondente à *feature* "Busca simples". O Quadro 5 descreve a variante "Busca avançada".

Nome: **CM002. Realizar busca avançada**
Atores: **Usuário final**
Fluxo normal: 1. Usuário informa "string" de busca e local onde deseja realizar a busca 2. Aplicação realiza busca no local especificado pelo usuário 3. Aplicação exibe lista com o conteúdo que corresponde à "string" informada 4. Usuário clica sobre um resultado e continua navegação
Fluxos alternativos: 3. Não existe conteúdo no local solicitado que seja associado à "string" informada 3.1. Aplicação exibe mensagem informando que não foi encontrado nenhum conteúdo no local especificado
Features relacionadas: **Navegação, Busca, Busca avançada**
Cenários afetados: **CM001. Realizar busca simples**

Quadro 5: Exemplo de caso de mudança para a *feature* alternativa de busca avançada

Neste exemplo, diferentemente do exemplo anterior, a intersecção entre os dois cenários é maior, pois este cenário descreve uma alternativa ao outro, e quase todos os mesmos passos são seguidos. Neste caso, recomenda-se fatorar os dois cenários, levando à criação de um terceiro que contenha as interações em comum, para facilitar o entendimento (ECKLUND JR; DELCAMBRE; FREILING, 1996). Neste exemplo, isto poderia ser atingido com a criação de um terceiro cenário **CM003. Exibir resultados da busca**, que condensa os passos 3 e 4 descritos no fluxo normal dos cenários anteriores, assim como os fluxos alternativos.

Em termos de formato, tanto casos de uso como casos de mudança são praticamente idênticos. A diferença é que, conforme descrito no próprio nome, um caso de mudança introduz uma alteração em um cenário que é considerado "normal", também chamado de base comum (COPLIEN, 2000). A definição do que é um comportamento "normal" em um domínio é altamente subjetiva, e escolhas diferentes podem levar a diferentes decisões posteriormente, durante o projeto do domínio.

Existem basicamente dois tipos de variabilidade em relação à base comum (COPLIEN, 2000):

- **Variabilidade positiva**: mantém a base comum intacta, não violando nenhuma de suas

premissas, enquanto novos comportamentos são adicionados ou refinados; e

- **Variabilidade negativa**: remove comportamento da base comum, no sentido em que uma ou mais premissas que eram válidas na base comum passam a ser violadas.

No exemplo do domínio web citado anteriormente, a busca avançada (Quadro 5) introduz uma variabilidade positiva no passo 1, pois este é um refinamento que acrescenta um comportamento adicional, não violando as premissas do cenário original. Também introduz uma combinação de variabilidades negativa e positiva no passo 2, pois neste cenário, não mais ocorre uma busca no conteúdo todo, apenas no conteúdo especificado pelo usuário. Os fluxos alternativos introduzem apenas variabilidades positivas, pois eles refinam o comportamento normal.

Em essência, e principalmente na fase de análise, ambos os tipos são válidos e ocorrem naturalmente dentro de um domínio. Porém, podem levar a decisões de projeto diferentes. Enquanto variabilidades positivas podem ser implementadas de maneira mais simples, as variabilidades negativas exigem mecanismos para suprimir comportamento da base comum (COPLIEN, 2000), o que pode levar a uma maior complexidade da arquitetura final. Assim, o uso de variabilidades positivas é preferido (COPLIEN; HOFFMAN; WEISS, 1998; COPLIEN, 2000).

Nesta tese, foram definidos os seguintes passos, numa forma sistemática, de se tomar a decisão sobre quais cenários fazem parte da base comum e quais introduzem variabilidades:

1. Começar com as *features* mandatórias, pois estas estarão presentes no domínio. Estas irão fazer parte dos casos de uso – os cenários que descrevem o comportamento "normal";

2. Para cada *feature* opcional, criar um caso de mudança;

3. Para as *features* alternativas onde é obrigatória a existência de ao menos uma variante, escolher aquela que representa a maioria dos casos para fazer parte da base comum. No exemplo acima, se a busca simples está presente em mais produtos do que a busca avançada, esta seria escolhida para inclusão como um caso de uso. A busca avançada seria modelada como caso de mudança; e

4. Caso o processo resulte em casos de mudança que apresentam variabilidade negativa, avaliar a possibilidade de transformação em variabilidade positiva, pois esta é preferível (COPLIEN; HOFFMAN; WEISS, 1998; COPLIEN, 2000).

Para realização do último passo, o seguinte método pode ser utilizado.

Supondo que um caso de uso UC001 possua 5 passos - p1, p2, p3, p4 e p5: após análise da variabilidade, é descoberto um caso de mudança CM001, que possui 4 passos: p1, p2, p4 e p5. CM001 introduz uma variabilidade negativa em UC001, pois ele remove p3 do comportamento "normal", ou base comum. Neste caso, não existe variabilidade positiva introduzida por CM001, e portanto basta inverter os cenários, transformando CM001 no cenário "normal", e UC001 no caso de mudança.

Porém, caso CM001 possua os passos p1, p2, p4, p5 e p6, a simples inversão não é suficiente, pois sempre existirá uma variabilidade negativa, não importando qual destes cenários faz parte da base comum. Neste caso, pode-se criar um terceiro cenário, que possui os passos em comum, por exemplo UC001: p1, p2, p4 e p5, e dois casos de mudança CM001: p1, p2, p3, p4 e p5 (antigo UC001), e CM002: p1, p2, p4, p5 e p6 (antigo CM001). Como resultado, somente variabilidades positivas são introduzidas.

O mesmo pode ser feito quando dois cenários diferem em um determinado passo. Neste caso, ocorre a combinação de variabilidade negativa e positiva (removendo um comportamento comum, e adicionando outro em substituição), e o mesmo método pode ser aplicado para remover a parte negativa.

No exemplo da busca avançada (Quadro 5), pode-se aplicar este método para remover a variabilidade negativa, resultando em 3 cenários:

1. Cenário 1: Busca
    (a) Usuário informa parâmetros de busca
    (b) Aplicação realiza busca no conteúdo conforme especificado pelos parâmetros
    (c) Aplicação exibe lista com o conteúdo que corresponde aos parâmetros especificados
    (d) Usuário clica sobre um resultado e continua navegando
2. Cenário 2: Busca simples
    (a) Usuário informa "string" de busca
    (b) Aplicação realiza busca em todo conteúdo
    (c) Aplicação exibe lista com o conteúdo que corresponde à "string" especificada
    (d) Usuário clica sobre um resultado e continua navegando
3. Cenário 3: Busca avançada
    (a) Usuário informa "string" de busca e local a ser buscado

(b) Aplicação realiza busca no local especificado

(c) Aplicação exibe lista com o conteúdo que corresponde à “string” e locais especificados

(d) Usuário clica sobre um resultado e continua navegando

Neste exemplo, as mudanças introduzidas pelos cenários 2 e 3 não invalidam o cenário 1, apenas acrescentam ou refinam comportamento, e portanto só introduzem variabilidades positivas.

A remoção forçada de uma variabilidade negativa, porém, pode interferir com o real significado dos modelos da base comum, produzindo impactos indesejados no projeto do domínio (COPLIEN, 2000). Por exemplo, em resposta aos cenários acima, um projetista pode projetar uma solução genérica onde o local a ser buscado é um parâmetro para o mecanismo de busca, que irá conter o nome deste local, ou a palavra “tudo”, caso se trate de uma busca simples. Em uma aplicação onde a busca avançada existe, esta solução é elegante e simples. Mas em uma aplicação onde a busca avançada não está presente, este parâmetro não faz sentido algum, e acrescenta uma sobrecarga tanto no processamento (para ler e interpretar o valor de entrada deste parâmetro), como na semântica dos modelos gerados.

Assim, antes de se aplicar este método, é necessário avaliar alguns fatores para determinar a sua real necessidade:

- **Peso da variabilidade negativa**: caso a variabilidade negativa ocorra com pouca frequência, pode não valer a pena gerar outros cenários apenas para removê-la. Por exemplo, caso apenas 1% das aplicações do domínio web possuam suporte para busca avançada, a solução descrita acima pode não valer a pena;

- **Tamanho da variabilidade negativa**: variabilidades negativas pequenas são mais facilmente implementadas, utilizando-se, por exemplo compilação condicional (COPLIEN, 2000; ANASTASOPOULOS; GACEK, 2001). Já a existência de variabilidades negativas grandes (envolvendo, por exemplo diversas classes ou cenários) pode indicar uma necessidade de reavaliação.

Atividade AD.3. Identificação de subdomínios

Papéis: analista do domínio, especialista do domínio

Entradas: *Informações sobre sistemas do domínio*, *Conhecimento do especialista*, *Informações sobre stakeholders*, *PT.1.Inicial. Planejamento do domínio*, *PT.2.Inicial. Mapa de aplicações*, *PT.3.Inicial. Modelagem do domínio*.

Saídas: *PT.4.Inicial. Candidatos a subdomínio*.

Descrição: a complexidade da maioria dos domínios torna virtualmente impossível a tarefa de se formalmente especificar o comportamento de um domínio completo (HADDAD; TESSER, 2002). Também prejudica a sua reusabilidade, uma vez que o escopo exato dos elementos do domínio e seus relacionamentos é difícil de compreender e gerenciar. Por estes motivos alguns autores, como Jarzabek (1997), defendem a idéia de que um domínio deve ser decomposto para facilitar o processo de desenvolvimento para reutilização.

Além disso, grandes sistemas dificilmente podem ser automatizados por um único gerador. Normalmente são necessários diferentes geradores trabalhando em conjunto para possibilitar a automação de um domínio completo (CZARNECKI; EISENECKER, 2000). Por estes motivos, na engenharia de domínio com a utilização de técnicas do MDD, a identificação dos subdomínios é uma tarefa crucial, pois permite ao engenheiro de domínio projetar e implementar ferramentas de modelagem, transformações e geradores específicos para os subdomínios que possuem maior capacidade de automação, de forma mais controlável.

Esta atividade lida com esta identificação dos subdomínios que podem ser automatizados utilizando técnicas do MDD, com foco no aumento da reusabilidade dos artefatos do domínio.

Existem na literatura poucos trabalhos que tratam da decomposição de um domínio em subdomínios (JARZABEK, 1997; HADDAD; TESSER, 2002; MAIDEN; SUTCLIFFE, 1996; LEE; KANG, 2003). Apesar de não almejarem a criação de DSLs voltadas para o desenvolvimento orientado a modelos, estes trabalhos descrevem problemas similares no contexto de reutilização, e oferecem algumas contribuições que podem auxiliar na execução desta atividade. Com base nesta literatura disponível, e nas experiências com estudos de casos, nesta tese as seguintes diretrizes foram definidas para esta atividade:

- *Foco na automação*: enquanto a maioria dos autores não apresenta critérios claros para decompor um domínio, no desenvolvimento orientado a modelos isto é muito claro: o subdomínio deve poder ser automatizado, através de ferramentas de modelagem e transformações;

- *Importância do especialista do domínio*: muitos autores concordam que o conhecimento do especialista do domínio é extremamente valioso nesta atividade (JARZABEK, 1997; HADDAD; TESSER, 2002; MAIDEN; SUTCLIFFE, 1996). Desta forma, a identificação de

subdomínios deve ser guiada por este profissional;

- *Dividir para conquistar*: o conceito básico de divisão de um problema grande em partes menores é também útil no processo de desenvolvimento para reutilização. Diferentes técnicas podem ser utilizadas, dependendo do contexto. Relacionamentos do tipo *ÉParteDe* e *ÉUm* podem ser utilizados como técnica de decomposição, onde cada subdomínio ou é parte de um subdomínio maior, ou uma instância. Além disso, a maioria dos autores concorda que a divisão deve seguir a categorização natural do domínio, o que mais uma vez ressalta a importância do especialista nesta tarefa;

- *Relacionamento features/sub-features*: *features* são divididas em sub-*features* para ajudar na tarefa de compreensão do domínio e sua variabilidade, e a análise destes relacionamentos (LEE; KANG, 2003) leva a uma sub-divisão natural do domínio. *Features* muito próximas são normalmente boas candidatas a pertencerem a um mesmo subdomínio. A atenção às *features* que aparentam estar separadas das demais pode também levar à identificação de um subdomínio distinto;

- *Domínios atômicos*: idealmente, o subdomínio identificado deve ser atômico, ou seja, não pode ser substancialmente decomposto sem alterar suas propriedades primárias (HADDAD; TESSER, 2002). Isto é importante para manter o subdomínio simples e gerenciável, e portanto mais fácil de automatizar; e

- *Repetição*: uma boa indicação sobre o que pode ser automatizado é o nível de repetição que ocorre com um projeto, estrutura ou trecho de código. Se algum trecho de código aparece repetidamente em diferentes partes do produto, mesmo que não de forma idêntica, é provável que uma máquina consiga fazer o trabalho de cópia parametrizada. Pode valer a pena tentar procurar um subdomínio associado a este trecho. Outra técnica para encontrar repetições é a busca por padrões recorrentes (KNODEL et al., 2005).

Além dessas diretrizes, a experiência prática com modelagem de domínio (TOLVANEN, 2005) mostra que um aumento incremental do escopo é a melhor forma de se chegar ao tamanho ideal do subdomínio, começando-se com um escopo pequeno, desenvolvendo partes da linguagem e transformações, e incrementando-o sucessivamente.

As seguintes sub-atividades são responsáveis pela identificação dos subdomínios.

Inicialmente, é feita uma seleção inicial de *candidatos a subdomínio (Sub-atividade AD.3.1)*. Em seguida, são *identificadas linguagens de modelagem (Sub-atividade AD.3.2)* e *ferramentas (Sub-atividade AD.3.3)*. Para cada subdomínio candidato, é feita a *atribuição*

de nível de confiança (Sub-atividade AD.3.4), onde são avaliadas a maturidade e o estado de cada subdomínio identificado até o momento. Por fim, os subdomínios são *documentados (Sub-atividade AD.3.5)*.

Sub-atividade AD.3.1. Seleção de candidatos a subdomínio

O objetivo desta sub-atividade é identificar potenciais subdomínios dentro do domínio. O analista do domínio tenta identificar, com base nas informações coletadas e com o auxílio das diretrizes descritas anteriormente, possíveis subdomínios. Apesar do conhecimento do especialista do domínio ser a principal fonte de informação para execução desta tarefa, o modelo de *features* pode ser útil. Observando-se o modelo de *features*, o analista pode identificar palavras-chave que representam um subdomínio. Conforme destacado por (MAIDEN; SUTCLIFFE, 1996), a categorização natural do domínio é a melhor indicação de onde encontrar um subdomínio.

Lee & Kang apresentam a técnica de análise de agrupamento de *features* (LEE; KANG, 2003), que pode ser utilizada nesta atividade. A técnica se baseia na identificação das unidades de agrupamento de *features* (FBU - *Feature Binding Units*). Uma FBU é um conjunto de *features* relacionadas que juntas executam um serviço ou operação dentro do domínio, ou seja, devem coexistir para que este serviço esteja presente no produto final. Esta técnica tem o objetivo de auxiliar no processo de configuração dos produtos, determinando quais conjuntos de *features* estarão presentes dependendo da configuração desejada.

O processo de análise das FBUs inicia-se com a identificação das *features* de capacitação que representam serviços independentemente configuráveis, ou seja, as macro funcionalidades principais do domínio que podem ser adicionadas ou removidas de um produto como unidades independentes. Normalmente, estas são as *features* localizadas no topo do diagrama. A partir destas *features* principais, percorre-se o modelo, incluindo as demais *features* relacionadas que são necessárias para que esta macro função possa ser executada.

Dentro de uma FBU identificada, podem existir *features* que são opcionais ou alternativas, podendo ser selecionadas de acordo com a necessidade do produto. Estes sub-conjuntos de *features* devem ser separados em FBUs diferentes, pois a exemplo das *features* de capacitação, também representam pontos de variação que podem ser independemente configuráveis.

Cada FBU é potencialmente um candidato a subdomínio, porém esta não é necessariamente a melhor escolha. Pode-se, por exemplo, unir duas ou mais FBUs em um único subdomínio, incluir ou remover *features* individualmente, ou mesmo sub-dividir uma FBU em mais de um subdomínio. Estas escolhas, porém, devem ser feitas somente mais adiante no processo, quando

mais maturidade sobre os subdomínios é adquirida.

Para cada candidato a subdomínio, as *features* correspondentes são identificadas. Isto pode ser realizado em uma matriz, relacionando cada *feature* ao seu subdomínio correspondente. Opcionalmente, pode ser produzida uma representação gráfica do subdomínio, em um diagrama que contém somente as *features* a ele pertencentes. A Figura 12 mostra quatro candidatos a subdomínio obtidos através da análise de FBUs do domínio de aplicações de autoria de conteúdo para Web.

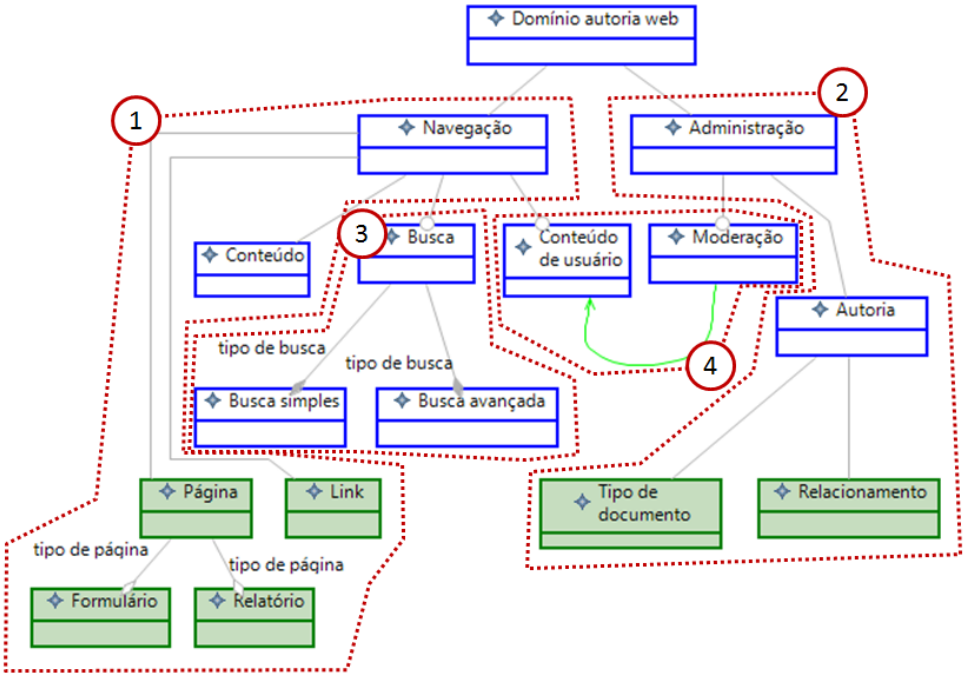


Figura 12: Candidatos a subdomínio do domínio web de autoria de conteúdo

Neste ponto, a sobreposição de subdomínios, caso ocorra, não é um problema, uma vez que pode ser que alguns dos subdomínios sejam descartados. Posteriormente, com a evolução do processo e os refinamentos que se seguem, este problema será analisado.

Sub-atividade AD.3.2. Identificação de linguagens de modelagem

O objetivo da identificação de subdomínios é possibilitar a definição de uma DSL que consiga representar a variabilidade não capturada nas *features* e nos cenários. Em domínios mais maduros, porém, podem já existir linguagens sendo utilizadas, como por exemplo a modelagem entidade-relacionamento, bastante comum. Mesmo que por algum motivo não possam ser diretamente utilizadas, essas linguagens podem oferecer pistas e informações importantes na definição de uma nova DSL. Nesta atividade, o analista do domínio tenta determinar se existe uma ou mais linguagens para o domínio. O especialista do domínio

é consultado, além das documentações existentes, tais como manuais e documentação das aplicações existentes, caso disponível.

A existência de uma linguagem específica de domínio é um dos indicativos de que este domínio está consideravelmente maduro e, portanto, os benefícios do desenvolvimento orientado a modelos serão maiores (TOLVANEN; KELLY, 2005).

Caso exista código-fonte disponível, há a possibilidade de existirem exemplos de modelos ou linguagens específicas para o domínio ou algum subdomínio.

É importante ressaltar que modelos nem sempre possuem uma representação gráfica que segue alguma linguagem conhecida, como a UML, por exemplo. Outras linguagens, incluindo arquivos de configuração e modelos textuais, devem também ser consideradas.

O modelo de *features* também pode oferecer dicas sobre o que procurar. Palavras-chave presentes no modelo de *features*, tais como nomes de *features*, podem ocorrer dentro da documentação ou código-fonte.

Após esta análise, é criada uma lista com as linguagens identificadas, associando-as com os subdomínios correspondentes.

Sub-atividade AD.3.3. Identificação de ferramentas

A exemplo das linguagens específicas de domínio, a existência de uma ferramenta de configuração ou modelagem é um indicativo de que o subdomínio em questão apresenta alto grau de maturidade, e portanto as chances da reutilização de software orientada a modelos ter sucesso são maiores. Em domínios extremamente maduros, podem existir até geradores de código que podem ser reaproveitados.

Novamente, o conhecimento do especialista do domínio é essencial, mas manuais e documentação também devem ser consultados, uma vez que eles podem referenciar ferramentas usadas para criar modelos ou gerar partes da aplicação. O código-fonte também deve ser inspecionado, pois é comum a existência, no código gerado, de dados sobre a ferramenta que o gerou, data da geração, entre outras informações úteis.

As ferramentas identificadas são então listadas e descritas, incluindo toda informação possível, uma descrição de suas funcionalidades, os artefatos que podem ser gerados por ela e referências para fontes externas. Novamente, caso seja encontrada uma ferramenta referente a um subdomínio ainda não identificado, acrescenta-se uma observação para posterior re-análise.

Sub-atividade AD.3.4. Atribuição de nível de confiança

Os subdomínios identificados nem sempre podem ser representados em uma DSL e automatizados utilizando técnicas do desenvolvimento orientado a modelos. Mesmo para aqueles que podem, o custo de se desenvolver linguagens, ferramentas e transformações pode ser muito alto. A automação de um subdomínio também pode requerer alterações e refinamentos no modelo de *features*, o que obviamente causa grande impacto no desenvolvimento. Finalmente, dependendo de qual subdomínio é implementado primeiro, outros subdomínios sobrepostos ou relacionados podem precisar ser revistos.

Por este motivo, todos os subdomínios identificados são tratados como candidatos até serem completamente realizados. Além disso, deve existir um certo nível de confiança de que um subdomínio irá render os resultados esperados quando realizado em forma de artefatos para MDD, antes de se iniciar o projeto e implementação. Nesse sentido, a medida de nível de confiança serve como uma ferramenta de gerenciamento de riscos, ajudando a garantir que mudanças críticas na arquitetura e nos modelos de análise levem aos resultados esperados. Também serve como um mecanismo de suporte à tomada de decisão, auxiliando na coordenação dos esforços durante o processo iterativo desta abordagem.

A determinação de um nível de confiança de um subdomínio é altamente subjetiva e dependente do conhecimento do especialista do domínio e da experiência dos engenheiros do domínio. Nesta tese, as seguintes questões foram identificadas por possuir impacto na decisão, e devem ser consideradas:

- Existe uma linguagem de modelagem para o subdomínio?
- Caso exista, qual é a maturidade desta linguagem: é uma linguagem bem conhecida, utilizada por especialistas em diferentes organizações? Existe somente na organização em questão? Foi desenvolvida somente para este projeto?
- A linguagem de modelagem foi validada, através de estudos de caso, para este projeto?
- Existe uma ferramenta específica para o subdomínio?
- Caso exista, qual é a maturidade desta ferramenta: é uma ferramenta bem conhecida, utilizada por especialistas em diferentes organizações? Existe somente na organização em questão? Foi desenvolvida somente para este projeto?
- A ferramenta foi validada, através de estudos de caso, para este projeto?

- Como a ferramenta se enquadra no projeto? Ela gera código executável? Se não, pode ser adaptada para gerar código? Quanto esforço é necessário para se utilizar a ferramenta neste projeto?
- Foi conduzido um projeto piloto para este subdomínio, utilizando-se uma linguagem de modelagem e ferramenta para geração de código?

Para determinar o nível de confiança de forma mais sistemática, o analista do domínio pode desenvolver uma métrica envolvendo estas e outras possíveis questões que possam surgir. Uma maneira simples é desenvolver um questionário com estas questões, atribuindo um peso para cada resposta. A soma de todos os valores é o nível de confiança.

O analista do domínio deve consultar todos os *stakeholders* quando definir esta métrica, uma vez que existem diferentes fatores a serem considerados. Dependendo do cenário, diferentes situações podem requerer valores diferentes. Por exemplo, para sistemas críticos, é razoável utilizar somente linguagens e ferramentas bem conhecidas, e portanto maiores pesos podem ser atribuídos para estas questões. Em projetos com prazos mais curtos de entrega, esta também pode ser a única opção. Porém, em projetos com mais tempo disponível, os valores podem ser ajustados para incluir mais subdomínios, já que há mais tempo para desenvolver ferramentas e linguagens de modelagem. Este é também o caso de domínios mais abrangentes.

Sub-atividade AD.3.5. Documentação dos candidatos a subdomínio

Nesta atividade, o analista do domínio documenta os subdomínios identificados, incluindo toda informação coletada nos passos anteriores: as *features* envolvidas, linguagens e ferramentas, e o nível de confiança. Na prática, a maior parte da documentação vai sendo construída durante o processo, e portanto esta atividade é realizada em paralelo com as anteriores.

Os Quadros 6 e 7 ilustram exemplos de documentação obtida nesta atividade. O Quadro 6 mostra a documentação do subdomínio de navegação, onde são informadas a descrição, as *features* envolvidas, linguagens e ferramentas identificadas. O Quadro 7 mostra a documentação do nível de confiança dos subdomínios identificados. Neste exemplo, somente quatro perguntas foram utilizadas, com peso atribuído conforme indicado no quadro.

Aqui o analista também descreve a interação entre o candidato a subdomínio e o restante do domínio. Neste ponto, esta descrição deve ser focada na cooperação em alto nível entre as *features*, com o objetivo de ajudar na decisão de investir na automação do subdomínio ou não. Em estágios posteriores, caso seja decidido que este subdomínio será automatizado, esta

Candidato a subdomínio: Navegação	
Descrição	Este sub-domínio envolve a navegação de uma aplicação web, com as páginas e links, e todo conteúdo visível ao usuário final.
Features	Navegação, Conteúdo, Página, Link, Formulário, Relatório
Linguagens de modelagem	
Diagrama de estados da UML	Uso de diagramas de estado para navegação, onde as páginas correspondem aos estados, e as transições correspondem aos links
UWE	Uso da UML e seus diagramas para modelagem dos diversos aspectos de uma aplicação Web (http://www.pst.ifi.lmu.de/projekte/uwe/)
WebML: hypertext model	Modelo hipertexto para especificação da navegação de uma aplicação web (www.webml.org)
WebML: Presentation model	Modelo para descrever as páginas de uma aplicação web (www.webml.org)
Ferramentas	
MagicDraw, ArgoUML, Jude	Ferramentas de modelagem UML que podem ser usadas para criar modelos UWE
Microsoft Visio	Ferramenta de criação de diagramas que pode ser estendida com formas específicas para WebML (www.webml.org)
MagicUWE	Plug-in UWE para a ferramenta MagicDraw (http://www.pst.ifi.lmu.de/projekte/uwe/toolMagicUWE.html)
ArgoUWE	Extensão da ferramenta ArgoUML para suporte ao UWE (http://www.pst.ifi.lmu.de/projekte/uwe/toolargoUWE.html)

Quadro 6: Documentação do subdomínio de navegação

interação deverá ser refinada, incluindo uma definição mais detalhada da interface entre os artefatos gerados para este subdomínio e outros artefatos.

Atividade AD.4. Validação e documentação do domínio

Papéis: analista do domínio, especialista do domínio

Entradas: *Informações sobre sistemas do domínio*, *Conhecimento do especialista*, *Informações sobre stakeholders*, *PT.1.Inicial. Planejamento do domínio*, *PT.2.Inicial. Mapa de aplicações*, *PT.3.Inicial. Modelagem do domínio*, *PT.3.Inicial. Candidatos a subdomínio*.

Saídas: *PT.1.Validado. Planejamento do domínio*, *PT.2.Validado. Mapa de aplicações*, *PT.3.Validado. Modelagem do domínio* e *PT.4.Validado. Candidatos a subdomínio*.

Descrição: antes do modelo do domínio ser utilizado, ele precisa ser validado e documentado. A documentação já foi iniciada durante as atividades anteriores. Nesta abordagem, as *features* são documentadas segundo o formato descrito por Czarnecki e Eisenecker (2000), que inclui a sua descrição semântica, o raciocínio, exemplo de aplicação, restrições, e outras informações (ALMEIDA et al., 2006).

Em seguida, é feita a validação do domínio. O analista de domínio verifica homônimos e sinônimos, com o objetivo de reduzir a ambiguidade. Em seguida, o domínio é validado

Atribuição de nível de confiança					
Foi definida uma métrica de confiança para avaliar os candidatos a sub-domínio, com base no seguinte questionário					
Questão					**Peso**
Q1. Existe uma linguagem de modelagem para o sub-domínio?					**1**
Q2. Existe uma ferramenta específica para o sub-domínio?					**2**
Q3. Caso exista, a ferramenta gera código conforme requerido pelo projeto?					**2**
Q4. Caso exista, a ferramenta é extensível, sendo possível modificar o suporte para linguagem e/ou geração de código?					**3**
Resultado					
Sub-domínio	**Q1**	**Q2**	**Q3**	**Q4**	**Nível de confiança**
Navegação	1	2	0	0	3
Busca	0	2	0	0	2
Cadastros	1	2	2	0	5
Conteúdo de usuário	0	0	0	0	0

Quadro 7: Documentação do nível de confiança dos subdomínios identificados

com relação às informações dos *stakeholders*, os requisitos iniciais e demais documentos relacionados.

Atividade AD.5. Decisão sobre inclusão/exclusão de subdomínios

Papéis: analista do domínio, especialista do domínio

Entradas: *Informações sobre sistemas do domínio*, *Conhecimento do especialista*, *Informações sobre stakeholders*, *PT.1.Validado. Planejamento do domínio*, *PT.2.Validado. Mapa de aplicações*, *PT.3.Validado. Modelagem do domínio*, *PT.4.Validado. Candidatos a subdomínio*.

Saídas: *PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios*.

Descrição: nesta atividade, o analista do domínio, junto com o especialista e os demais *stakeholders*, analisam os subdomínios identificados com o objetivo de determinar se os mesmos serão incluídos nas fases subsequentes de projeto e implementação.

Essa decisão deve levar em consideração o nível de confiança dos subdomínios candidatos, e portanto a existência de linguagens, ferramentas e outros indícios de sua maturidade, seus relacionamentos com os demais elementos do domínio, e outros fatores externos, incluindo os objetivos de negócio e condições de mercado.

A abordagem baseia-se num processo iterativo, e alguns subdomínios podem estar mais

maduros, enquanto outros necessitam de mais desenvolvimento. Neste sentido, ao invés de simplesmente incluir ou excluir um subdomínio, existem diferentes níveis de decisões (D) a serem tomadas:

- D_1. **Exclusão imediata**: significa que o candidato a subdomínio não é passível de automação, e deve ser descartado imediatamente. Tipicamente, este subdomínio tem um baixo nível de confiança, não existem linguagens de modelagem ou ferramentas que dão suporte;

- D_2. **Manter para avaliação posterior**: significa que este subdomínio tem chance de ser automatizado, mas não existe evidência concreta de que isto pode ser feito. Tipicamente, tem um baixo nível de confiança, nenhuma linguagem de modelagem ou ferramenta, mas o especialista do domínio, ou a experiência com o desenvolvimento, indicam que ele pode ser útil após algum desenvolvimento. Não existe maneira de se estimar o esforço para torná-lo um subdomínio concreto, ou seja, é muito arriscado iniciar algum desenvolvimento neste sentido. Portanto, essa decisão indica que nenhuma ação será tomada nesta iteração. No entanto, caso os *stakeholders* aceitem o risco, este mesmo candidato a subdomínio pode se enquadrar no próximo nível de decisão (D_3);

- D_3. **Iniciar investigação**: se um subdomínio possui potencial para ser automatizado, mas as ferramentas para isso não estão disponíveis, ele pode ser investigado, por meio do desenvolvimento de protótipos de linguagens de modelagem e/ou ferramentas de modelagem e transformações. Apesar de esta decisão levar à alocação de recursos e esforços, estes são mais limitados, uma vez que não há impacto nos demais artefatos do domínio, e é sempre possível parar a investigação sempre que necessário, pois o sucesso do negócio não depende diretamente desse desenvolvimento;

- D_4. **Iniciar o desenvolvimento de artefatos de produção**: este é o ponto sem retorno para um subdomínio. Neste nível de decisão, a organização compromete-se a incluir o subdomínio no processo de desenvolvimento, diferentemente do nível D_3, onde ainda existe a possibilidade de se descartar o subdomínio. Um subdomínio neste nível deve possuir um alto nível de confiança, mas não necessariamente precisa possuir as ferramentas para ser automatizado. Esta é uma decisão crítica, uma vez que significa empregar esforços e recursos de forma mais intensa, para automatizar o subdomínio e gerenciar o seu impacto no restante do domínio;

- D_5. **Iniciar um projeto piloto**: antes de incluir um subdomínio no processo final, é boa prática executar um projeto piloto, para reduzir os riscos da introdução de uma nova

tecnologia no desenvolvimento, e reduzir as barreiras à transferência tecnológica que irá ocorrer. Este projeto piloto também serve para se verificar os benefícios reais da nova tecnologia, e planejar a melhor maneira de aplicá-la a projetos reais; e

- D_6. **Inclusão imediata**: significa que o subdomínio está maduro o suficiente, possui ferramentas e uma ou mais linguagens de modelagem estáveis e validadas. O nível de confiança é alto, e o subdomínio já pode ser incluído nas fases de projeto e implementação. Também significa que o desenvolvimento dos artefatos deste subdomínio é guiado pelas linguagens e/ou ferramentas definidas. Enquanto a maioria dos subdomínios somente irá alcançar este nível após passar pelos níveis 2, 3, 4 e 5, alguns subdomínios bem conhecidos podem começar diretamente no nível 5. Alguns podem até começar diretamente no nível 6, se a organização sentir que possui a experiência necessária para lidar com esta tecnologia.

Para tomar a decisão correta, o analista do domínio deve considerar toda informação disponível, e a decisão deve ser acordada com todos os *stakeholders*. Além da informação já mencionada, alguns outros critérios podem ajudar:

- **Experiência da equipe de desenvolvimento:** se a equipe de desenvolvimento possui experiência com algum subdomínio, pode ser vantajoso focar neste subdomínio, para que seus benefícios possam ser alcançados sem muito esforço de entendimento e aprendizado;

- **Tamanho do subdomínio:** sempre que possível, deve-se começar com subdomínios menores e aumentá-los gradativamente (TOLVANEN, 2005). Apesar de domínios menores levarem a menos benefícios em termos de reutilização, são mais fáceis de gerenciar e implementar. Com o tempo, eles tendem a evoluir;

- **Nível de abstração:** subdomínios que se encontram em um nível de abstração próximo ao código-fonte são mais simples de implementar, e estão entre os exemplos de maior sucesso no desenvolvimento orientado a modelos, em termos de facilidade de geração de código (TOLVANEN; KELLY, 2005). Em contrapartida, são em geral menos reaproveitáveis; e

- **Existência de código-exemplo:** a existência de exemplos de como o código ou outros artefatos devem ser gerados é um fator importante. A abordagem de desenvolvimento através de exemplos (WIMMER et al., 2007) facilita o processo de construção de transformadores e geradores de código.

Estes são apenas exemplos de critérios que podem ser considerados durante a tomada de decisão. Cada domínio pode introduzir seus próprios conjuntos de critérios e particularidades a serem considerados.

Como referência, pode-se também utilizar a métrica de nível de confiança, definindo-se intervalos de valores para cada decisão. Por exemplo, em uma escala de 0 a 60, subdomínios com nível de confiança entre 0 e 10 levam a decisão D_1. Entre 11 e 20, decisão D_2, e assim por diante. Porém, esta técnica serve apenas como referência. A decisão final deve ser tomada levando-se em conta outros fatores, e não somente os incluídos na métrica.

Os níveis de decisão são alterados à medida que o domínio evolui durante as diversas iterações do seu ciclo de vida, conforme descrito na Seção 4.4. À medida que novos desenvolvimentos acontecem, novas informações surgem como resultado e passam a ser consideradas durante a atividade de decisão. A Figura 13 mostra uma possível evolução dos níveis de decisão referentes a quatro candidatos a subdomínios do domínio de aplicações de autoria de conteúdo para Web.

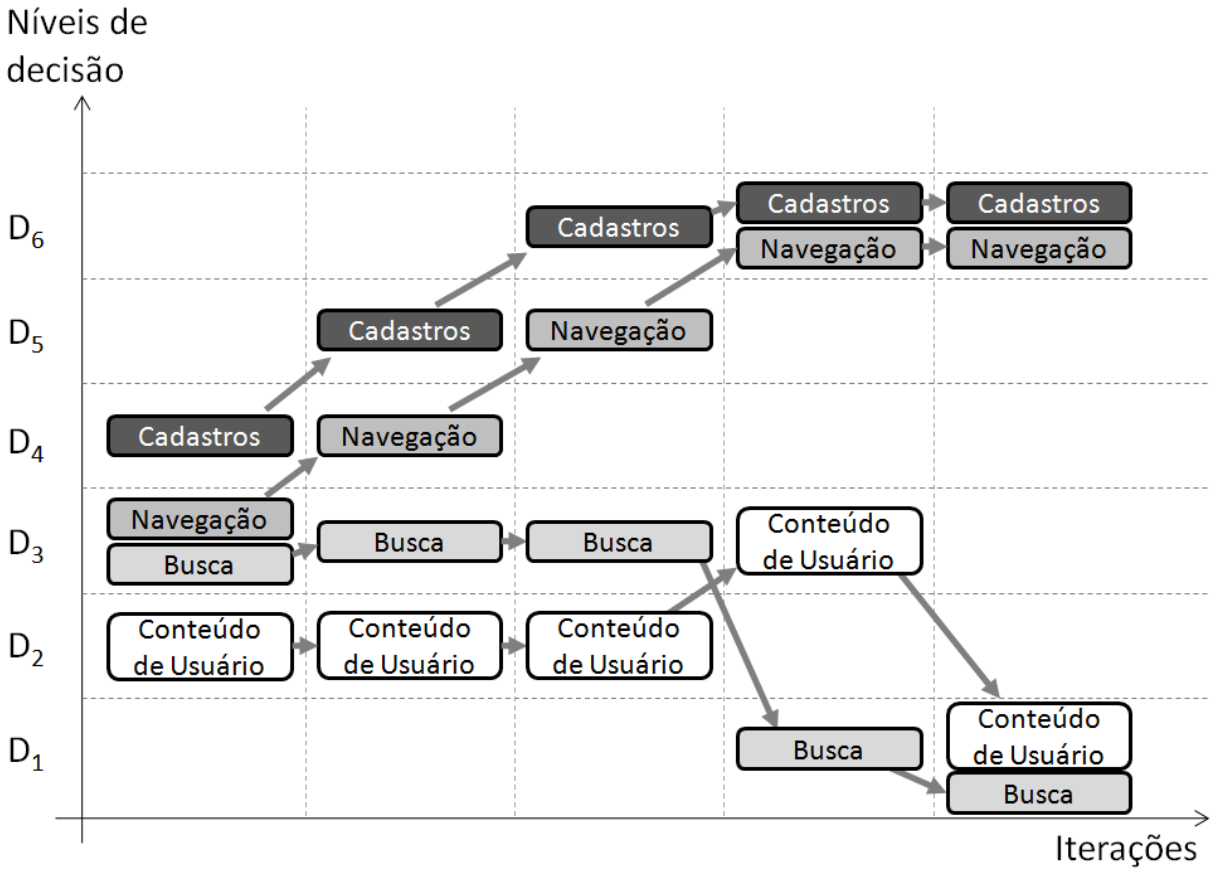


Figura 13: Possível evolução dos níveis de decisão para o domínio de aplicações de autoria de conteúdo para Web

Inicialmente, toma-se uma decisão com base nas informações disponíveis inicialmente.

Neste caso, o subdomínio de **cadastros** apresentou um alto grau de confiança, por já existirem linguagens e ferramentas disponíveis, e que geram código conforme requerido pelo projeto. Por isso foi colocado no nível D_4, onde inicia-se o desenvolvimento dos artefatos de produção. Nas iterações seguintes, esse subdomínio passou para o nível D_5, onde iniciou-se um projeto piloto, e finalmente D_6, onde o mesmo passou a fazer parte do domínio.

Os subdomínios de **navegação** e **busca** foram inicialmente colocados no nível D_3, ficando sob investigação, pois a confiança sobre seu sucesso dentro do domínio não era suficiente. Após algumas iterações, porém, o subdomínio de **navegação** foi considerado confiável o suficiente para iniciar a produção, que culminou com sua passagem para o nível D_6. Já o subdomínio de **busca**, após investigação, foi considerado excluído, passando para o nível D_1.

O subdomínio de **conteúdo de usuário** apresentou um baixo nível de confiança inicial, e portanto permaneceu no nível D_2 para investigação posterior. Assim que todos os outros subdomínios terem sido investigados e terem sua situação definida, este subdomínio entrou em investigação, no nível D_3. Mas após investigação, concluiu-se que o mesmo também deveria ser excluído, terminando no nível D_1. Ao final dessa evolução, o domínio passa a contar com os subdomínios de **cadastros** e **navegação**.

Após a tomada de decisão para todos os candidatos a subdomínio, o analista do domínio deve lidar com a sobreposição entre eles. Neste ponto, só é possível determinar se dois subdomínios irão interferir um com o outro analisando se eles possuem *features* em comum. Mesmo que não existam *features* em comum, pode ser que os artefatos gerados para um subdomínio interfiram ou tenham impacto nos artefatos de outro subdomínio. E não é possível determinar isto sem aprofundar-se no projeto e implementação.

Além disso, mesmo que não exista sobreposição de subdomínios em nenhum nível, ainda assim a automação de um subdomínio pode afetar outros. Por exemplo, a automação de um determinado subdomínio pode levar a mudanças/refinamentos nos requisitos, *features* ou casos de uso, para dar suporte a uma nova linguagem de modelagem ou ferramenta. Dessa forma, outros subdomínios dependentes dos mesmos requisitos, *features* ou casos de uso serão afetados.

Para evitar este problema, a seguinte restrição pode ser aplicada durante a tomada de decisões: *somente um subdomínio pode estar em desenvolvimento a cada vez. Isto significa que as decisões D_4, D_5 e D_6 são mutuamente exclusivas. Se já existir um subdomínio sendo desenvolvido ou testado em um projeto piloto, outros não poderão estar na mesma situação.*

Isto não vale para a decisão D_3, uma vez que não há problemas em se investigar um subdomínio junto com o desenvolvimento de outro. Podem inclusive existir múltiplos

subdomínios sendo investigados ao mesmo tempo.

Se uma organização considerar esta restrição muito rígida, pode optar por ignorá-la. Porém, ela deve estar atenta às possíveis sobreposições ou interferências entre dois subdomínios sendo colocados em produção ao mesmo tempo. Na prática, em geral, esta situação não ocorrerá com muita frequência, já que não é comum a existência de muitos subdomínios sendo automatizados.

5.3 Considerações finais

Neste capítulo foi apresentada a fase de análise de domínio, com atividades para promover a reutilização através do desenvolvimento orientado a modelos. As principais contribuições deste capítulo são as atividades sistemáticas e algumas diretrizes para identificação dos subdomínios para automação. Também apresenta uma forma para lidar com os riscos da incerteza sobre a automação dos subdomínios.

O Quadro 8 resume as atividades da análise de domínio.

Atividades e sub-atividades	**Entradas**	**Saídas**
AD.1. Planejamento do domínio AD.1.1. Preparação AD.1.2. Definição do escopo	Informações sobre sistemas do domínio Conhecimento do especialista Informações sobre *stakeholders*	PT.1.Inicial. Planejamento do domínio PT.2.Inicial. Mapa de aplicações
AD.2. Modelagem do domínio AD.2.1. Mapeamento da variabilidade em cenários	Informações sobre sistemas do domínio Conhecimento do especialista Informações sobre *stakeholders* PT.1.Inicial. Planejamento do domínio PT.2.Inicial. Mapa de aplicações	PT.3.Inicial. Modelagem do domínio
AD.3. Identificação de subdomínios AD.3.1. Seleção de candidatos a subdomínio AD.3.2. Identificação de linguagens de modelagem AD.3.3. Identificação de ferramentas AD.3.4. Atribuição de nível de confiança AD.3.5. Documentação dos subdomínios candidatos	Informações sobre sistemas do domínio Conhecimento do especialista Informações sobre *stakeholders* PT.1.Inicial. Planejamento do domínio PT.2.Inicial. Mapa de aplicações PT.3.Inicial. Modelagem do domínio	PT.4.Inicial. Candidatos a subdomínio
AD.4. Validação e documentação do domínio	Informações sobre sistemas do domínio Conhecimento do especialista Informações sobre *stakeholders* PT.1.Inicial. Planejamento do domínio PT.2.Inicial. Mapa de aplicações PT.3.Inicial. Modelagem do domínio PT.4.Inicial. Candidatos a subdomínio	PT.1.Validado. Planejamento do domínio PT.2.Validado. Mapa de aplicações PT.3.Validado. Modelagem do domínio PT.4.Validado. Candidatos a subdomínio
AD.5. Decisão sobre inclusão/exclusão de subdomínios	Informações sobre sistemas do domínio Conhecimento do especialista Informações sobre *stakeholders* PT.1.Validado. Planejamento do domínio PT.2.Validado. Mapa de aplicações PT.3.Validado. Modelagem do domínio PT.4.Validado. Candidatos a subdomínio	PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios

Quadro 8: Resumo das atividades para análise de domínio orientada a modelos

O Quadro 9 descreve os produtos de trabalho da fase da análise de domínio.

O produto da análise de domínio é a definição completa do escopo, e a modelagem das

Produto de trabalho	**Descrição**	**Estado**
PT.1. Planejamento do domínio	Documento que descreve o domínio, seus objetivos e restrições, além de listar os *stakeholders* e as aplicações de exemplo	1. *Inicial*: versão inicial do planejamento, pode conter inconsistências. 2. *Validado*: planejamento verificado em busca de inconsistências e erros.
PT.2. Mapa de aplicações	Documento que mapeia todas as aplicações do domínio e suas respectivas *features*	1. *Inicial*: versão inicial do mapa, pode conter inconsistências. 2. *Validado*: mapa verificado em busca de inconsistências e erros.
PT.3. Modelagem do domínio	Documento que detalha as *features* do domínio, especificando seus tipos e relacionamentos. Também contém os casos de uso do domínio, e informações sobre os subdomínios	1. *Inicial*: versão inicial da modelagem, pode conter inconsistências. 2. *Validado*: modelagem verificada em busca de inconsistências e erros.
PT.4. Candidatos a subdomínio	Documento listando os subdomínios, as *features* correspondentes, e as listas de linguagens de modelagens e ferramentas específicas para os subdomínios	1. *Inicial*: versão inicial do documento, pode conter inconsistências. 2. *Validado*: documento verificado em busca de inconsistências e erros.
PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios	Documento que registra as decisões sobre inclusão/exclusão dos subdomínios nas diferentes iterações do processo	Nenhum

Quadro 9: Descrição dos produtos de trabalho da fase de análise de domínio

variabilidades e comunalidades, tanto em termos de *features*, que descrevem as variabilidades estáticas, como em termos de cenários, que descrevem as variações no comportamento. Também são definidos nesta fase os candidatos a subdomínios automatizáveis.

6 *Projeto de domínio orientado a modelos*

O projeto do domínio é uma fase essencial da engenharia de domínio (BOSCH, 2000). Seu objetivo é definir uma arquitetura de software específica de domínio, e um conjunto de artefatos reutilizáveis que podem ser combinados para desenvolver aplicativos mais rapidamente e com maior qualidade.

Uma arquitetura de software específica de domínio é uma combinação de componentes de software, especializada para um tipo de tarefa em particular (domínio), generalizada para uso efetivo neste domínio e composta em uma estrutura padronizada efetiva para a construção de aplicações bem sucedidas (TRACZ, 1995). Em geral, a arquitetura deve ser projetada para atender aos principais requisitos conforme percebidos pelos *stakeholders*, de forma a alcançar os principais atributos de qualidade que são importantes para uma situação em particular (BASS; CLEMENTS; KAZMAN, 2003).

Em termos de engenharia de domínio, um dos pontos principais na definição da arquitetura do domínio é a reutilização de software, e a capacidade de se desenvolver produtos de software similares rapidamente, sem muitas modificações ou adaptações nessa arquitetura. Isto pode ser alcançado por meio de um bom suporte à comunalidade e variabilidade do domínio (MOON; YEOM, 2006; CZARNECKI, 2005; WEISS; LAI, 1999; CLEMENTS; NORTHROP, 2002). A arquitetura projetada deve não somente prever os pontos de variação, mas também efetivamente providenciar o suporte requerido para sua implementação (BASS; CLEMENTS; KAZMAN, 2003).

Há dois tipos fundamentalmente diferentes de complexidade relativa à variabilidade (VÖLTER; GROHER, 2007; CZARNECKI, 2005):

- **Configuração de rotina**: tipo de variabilidade em que estruturas simples, como uma lista de propriedades ou uma árvore, podem ser utilizados para configuração do produto; e
- **Construção criativa**: tipo de variabilidade em que são necessários modelos complexos, como estruturas do tipo grafo ou linguagens textuais completas, para descrevê-la

completamente.

Diferentes mecanismos de representação de variabilidade podem estar em diferentes locais de um espectro que vai desde a configuração de rotina até a construção criativa. Por exemplo, *wizards* são mecanismos simples, onde a cada passo um parâmetro é especificado, e portanto localizam-se próximo à configuração de rotina. O modelo de *features* (Seção 3.1.1) é um pouco mais complexo, mas ainda é relativamente simples, localizando-se aproximadamente na metade do espectro. Do lado criativo, encontram-se as linguagens específicas de domínio (DSL) (CZARNECKI, 2005).

O modelo de *features* do domínio de autoria de conteúdo para Web, já utilizado no capítulo anterior e replicado aqui na Figura 14, contém duas *features* principais: navegação e administração. A navegação (*feature* mandatória) consiste no conteúdo visível ao usuário e busca automática, que pode ser simples e/ou avançada (*features* alternativas do tipo *or*, onde mais de uma alternativa pode estar presente). A submissão de conteúdo de usuário é opcional. No lado da administração, a *feature* de autoria (mandatória) representa as funções de publicação do conteúdo. Se existir conteúdo de usuário, este pode ou não ser moderado (a seta curva indica uma dependência entre moderação e conteúdo de usuário). Estas são *features* de capacitação (Seção 3.1.1).

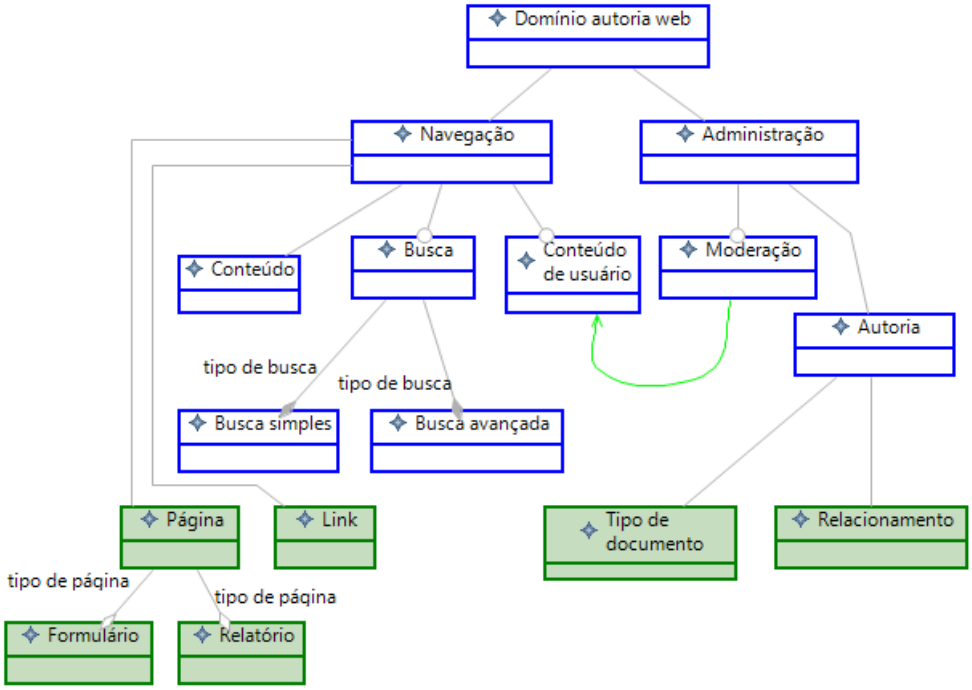


Figura 14: Modelo de *features* do domínio web de autoria de conteúdo

O que também varia de aplicação para aplicação deste domínio é a estrutura da informação (tipos de documento e seus relacionamentos) e como a mesma é apresentada ao usuário (páginas

e links). Para representar esta variabilidade, no lado inferior da figura estão as *features* de tecnologia do domínio (Seção 3.1.1): páginas (formulários ou relatórios) e *links* são utilizados para exibir a informação em um navegador, e tipos de documentos e relacionamentos descrevem a estrutura da informação.

A maioria das *features*, como a busca, conteúdo de usuário e moderação, representa a variabilidade do tipo presente/ausente, que fica próxima do lado da configuração de rotina no espectro de variabilidade. Desta forma, grande parte do suporte à variabilidade pode ser conseguido somente através de configuração, com a ajuda de uma arquitetura bem projetada e um conjunto de componentes reutilizáveis.

Porém, *features* como tipos de documentos e relacionamentos contêm variabilidades mais complexas, para as quais todos os detalhes não podem ser capturados somente com modelos de *features*, pois estes são muito genéricos (TOLVANEN; KELLY, 2005). Estruturas ou mecanismos mais ricos são necessários, com mais poder expressivo capaz de capturar detalhes sobre os conceitos do domínio, seus relacionamentos e restrições.

Nestes casos, é possível simplesmente implementar manualmente o suporte a esta variação, que é o que acontece no desenvolvimento tradicional. Para isso, um desenvolvedor utiliza suas habilidades de programador e uma linguagem de programação, possivelmente com a ajuda de uma biblioteca ou *framework* que facilite este trabalho criativo.

Mas no caso do MDD, onde se deseja automatizar o suporte à variabilidade, a tecnologia escolhida é normalmente uma linguagem específica de domínio (DSL), pois ela pode complementar o modelo de *features* com detalhes sobre variabilidades mais complexas em partes do domínio (TOLVANEN; KELLY, 2005; CZARNECKI, 2005). Adicionalmente, uma DSL pode ser utilizada junto com transformações para automaticamente gerar artefatos de implementação, que é o objetivo final do desenvolvimento orientado a modelos (CZARNECKI, 2005).

O elemento que une estes artefatos é a arquitetura do domínio, que deve ser projetada para dar suporte tanto à variabilidade baseada em *features* como à variabilidade baseada em DSLs. Também deve ter preocupações sobre a existência, além dos componentes do domínio, de artefatos específicos do MDD, como DSLs, transformações de software e geradores de código.

Um problema é que a maioria das abordagens de engenharia de domínio existentes não define como projetar uma arquitetura de software específica de domínio com este objetivo. A literatura possui muitos trabalhos que discutem detalhes sobre como produzir uma arquitetura para reutilização (BASS; CLEMENTS; KAZMAN, 2003; BOSCH, 2000), mas estes não consideram o MDD. E aqueles que consideram são normalmente focados no processo de derivação automática

de produtos (WEISS et al., 2008; VÖLTER; GROHER, 2007; BOTTERWECK; O'BRIEN; THIEL, 2007; ARBOLEDA; CASALLAS; ROYER, 2007; TRASK et al., 2006; TOLVANEN; KELLY, 2005; CZARNECKI; HELSEN; EISENECKER, 2004b), e não na tarefa de produzir uma arquitetura de domínio que é adequada para o MDD, ou seja, que contemple elementos capazes de integrar DSLs, transformações e geradores de código.

Além disso, a maioria destas abordagens, como as descritas por Almeida et al. (2007b), Keepence e Mannion (1999), Lee e Kang (2004), baseia-se na identificação da variabilidade somente em termos de *features*. Como já discutido anteriormente, há outros tipos de variação, como aquelas presentes em modelos entidade-relacionamento, por exemplo (BARTHOLDT; OBERHAUSER; RYTINA, 2008), que são mais complexas do que é possível capturar em um modelo de *features* (RABISER; GRÜNBACHER; DHUNGANA, 2007), sendo necessária uma DSL específica para representar a variabilidade de modo adequado ao MDD.

6.1 Objetivos do projeto de domínio

Existem conceitos relacionados ao MDD que devem ser considerados durante o projeto do domínio, especialmente durante a definição da arquitetura do domínio. Além da variabilidade mais complexa discutida anteriormente, nesta abordagem a arquitetura e seus componentes contam com a existência de DSLs e transformações, que por sua vez devem ser capazes de produzir artefatos de implementação voltados àquela arquitetura específica.

A fase de projeto do domínio tem os seguintes objetivos:

- **Identificar as diretrizes arquiteturais**: o projeto arquitetural deve ser direcionado pelos objetivos de negócio e requisitos mais importantes, considerando-se o ponto de vista de diferentes *stakeholders*. Estes são chamados de **diretrizes arquiteturais** (BASS; CLEMENTS; KAZMAN, 2003), e são responsáveis por "moldar" a arquitetura de forma a atender a todos os requisitos da melhor forma possível. Nos contextos de reutilização e MDD, as diretrizes devem incluir, obrigatoriamente, as variabilidades em forma de *features* e DSLs, e a existência de artefatos do MDD, como DSLs e transformações de software;

- **Selecionar/definir táticas e padrões**: o projeto arquitetural deve cobrir as diretrizes arquiteturais, em forma de táticas e padrões que contemplem soluções para cada requisito. Um padrão consiste em uma solução comprovadamente bem sucedida para determinados tipos de problema, com um contexto bem definido. Por trás de um padrão existem

uma ou mais táticas, que representam as maneiras utilizadas pelo padrão para resolver o problema;

- **Definir a arquitetura e componentes**: o projeto do domínio deve incluir uma definição da arquitetura do domínio. Junto com a arquitetura, devem ser especificados os componentes do domínio, incluindo componentes de software tradicionais, serviços, e também artefatos do MDD, como DSLs, transformações e geradores de código; e

- **Documentar a arquitetura**: como objetivo desta fase, deve ser produzida toda a documentação referente à arquitetura projetada, visando servir de guia para a fase de implementação.

Esta fase de projeto é inspirada nos métodos descritos por Almeida et al. (2007b), deBaud e Schmid (1999), Bass, Clements e Kazman (2003), Clements, Kazman e Klein (2004), com duas diferenças principais:

1. São oferecidos meios mais concretos para se elicitar diretrizes arquiteturais explícitas para o suporte aos diferentes tipos de variabilidade, na forma de *features*, cenários e linguagens específicas de domínio, o que não é descrito nos métodos citados; e

2. Juntamente com as diretrizes, são providenciadas táticas para atendê-las, utilizando padrões de projeto específicos para as diferentes formas de variabilidade, a exemplo de outras abordagens da literatura (ALMEIDA et al., 2007b; KEEPENCE; MANNION, 1999; LEE; KANG, 2004).

6.2 Atividades do projeto de domínio

O projeto do domínio é um processo iterativo de refinamento. Inicia-se com o domínio todo, e a cada iteração é feito um refinamento que identifica novos módulos, que serão refinados na próxima iteração, e assim por diante, até que o projeto esteja concluído, em função de um entendimento satisfatório sobre o que deverá ser implementado. A primeira atividade cuida da escolha dos módulos a serem refinados (Atividade PD.1), sendo executada no início de cada iteração. Em seguida, são selecionadas as diretrizes arquiteturais (Atividade PD.2), que são os requisitos mais importantes a serem atendidos pelo projeto da arquitetura. Para cada diretriz, escolhe-se um ou mais padrões arquiteturais (Atividade PD.3), responsáveis por resolver cada requisito individualmente, ajudando na obtenção dos atributos de qualidade desejados para a arquitetura. Os padrões são então aplicados durante o refinamento dos módulos (Atividade

PD.4), de forma a identificar novos módulos e dar "forma" à arquitetura. Por fim, uma atividade avalia a arquitetura ou arquiteturas projetadas, caso sejam definidas diferentes opções arquiteturais (Atividade PD.5).

Estas atividades são detalhadas a seguir.

Atividade PD.1. Escolha dos módulos a serem refinados

Papéis: projetista do domínio

Entradas: *PT.3.Validado. Modelagem do domínio*, *PT.4.Validado. Candidatos a subdomínio* e *PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios*

Saídas: *PT.6. Módulos a serem refinados*

Descrição: o projeto arquitetural desta abordagem baseia-se no método ADD (*Attribute-Driven Design* ou projeto orientado a atributos (BASS; CLEMENTS; KAZMAN, 2003)). O ADD promove o refinamento sucessivo de módulos, iniciando-se com o domínio todo, e realizando divisões até serem obtidos os módulos que irão formar a arquitetural final.

Assim, esta primeira atividade consiste na escolha dos módulos a serem refinados. O refinamento pode ocorrer em duas dimensões:

- **Dos pontos comuns para os variáveis**: neste refinamento, inicia-se o projeto considerando-se apenas os pontos comuns. Em seguida, a cada iteração, acrescenta-se um ponto de variação no projeto; e
- **De módulos para sub-módulos**: neste refinamento, inicia-se dividindo-se o domínio todo em módulos, e em seguida os módulos são subdivididos sucessivamente.

Estas dimensões normalmente são ortogonais, ou seja, um módulo pode incluir diversos pontos variáveis. Da mesma forma, um ponto variável pode incluir diversos módulos. Portanto, recomenda-se realizar apenas um desses tipos de refinamento a cada iteração.

Não existe uma ordem específica, ou seja, pode-se iniciar com um refinamento de módulo para sub-módulo, em seguida acrescentar um ponto variável, depois novamente refinar um módulo. Porém, sugere-se iniciar com uma primeira divisão do domínio em módulos, o que permite uma visão geral da arquitetura desejada.

Após uma primeira divisão, o projetista do domínio avalia a arquitetura existente até o momento, identifica se existem módulos que ainda precisam ser refinados, e os lista. Para cada um destes, executa as atividades seguintes, refinando o módulo novamente.

O refinamento segue até o projetista decidir que não é necessário acrescentar mais detalhes. A arquitetura deve capturar apenas as informações mais importantes do projeto, e portanto não são necessários muitos detalhes. Bass, Clements e Kazman (2003), após anos de experiência em projetos envolvendo projeto arquitetural, observaram que três níveis de divisão são normalmente suficientes, pois permitem detalhar de forma satisfatória as informações essenciais para que o processo possa seguir adiante.

É importante lembrar que nesta abordagem já existe uma primeira divisão do domínio em subdomínios, feita na atividade de análise. Porém, esta é uma divisão conceitual que pode não coincidir com a divisão em módulos que é realizada nesta fase de projeto (BUSCHMANN et al., 1996). Enquanto a primeira tem como objetivo identificar partes do domínio onde a automação é possível, separando conjuntos de *features* relacionadas, aqui a divisão tem como objetivo implementar padrões arquiteturais que irão atender a requisitos para o domínio – as diretrizes arquiteturais.

Atividade PD.2. Seleção das diretrizes arquiteturais

Papéis: projetista do domínio, especialista do domínio, demais *stakeholders*.

Entradas: *PT.6. Módulos a serem refinados*

Saídas: *PT.7. Diretrizes arquiteturais*

Descrição: as diretrizes arquiteturais são uma combinação de requisitos funcionais e de qualidade que dão "forma" à arquitetura de um domínio ou módulo em particular (BASS; CLEMENTS; KAZMAN, 2003). As diretrizes são normalmente representadas através de cenários que testam a capacidade da arquitetura em satisfazer um ou mais atributos de qualidade.

Para identificar as diretrizes, os objetivos de negócio mais importantes são identificados e transformados em cenários ou casos de uso. Desta lista, aqueles com maior impacto na arquitetura são escolhidos. Estas são as diretrizes arquiteturais (BASS; CLEMENTS; KAZMAN, 2003).

Nesta abordagem, ao menos três tipos de diretrizes devem estar presentes com maior prioridade:

- **Variabilidade em termos de *features***: como discutido anteriormente, a maior parte da variabilidade pode ser descrita através de *features*. Para o projeto arquitetural, a variabilidade é descrita também na forma de cenários, visando facilitar seu entendimento e futura avaliação;

- **Variabilidade em forma de DSLs**: casos mais complexos de variabilidade exigem um mecanismo mais expressivo para expressá-la apropriadamente. Para o projeto arquitetural, são utilizadas DSLs para descrever esta variabilidade mais complexa; e

- **Integração entre subdomínios**: o projeto arquitetural precisa considerar os pontos onde diferentes subdomínios irão interagir. Isto envolve, por exemplo, a integração entre código gerado e não-gerado. Estes pontos de interação devem ser explícitos para que a arquitetura possa dar suporte adequado à geração de código.

Sub-atividade PD.2.1. Seleção das diretrizes arquiteturais de variabilidade baseada em *features*

Nesta atividade, o objetivo é descrever cenários que ilustrem as variabilidades que ocorrem em termos da presença ou ausência de *features*. Tais cenários já foram identificados, durante a análise, na atividade de mapeamento das *features* em forma de cenários (Sub-atividade AD.2.1).

Portanto, aqui é necessário somente selecionar, dentre os cenários descritos na análise, aqueles que descrevem a variabilidade que diz respeito ao módulo sendo refinado na iteração atual. Caso seja necessário, pode-se enriquecer estes cenários com mais informações, tais como atributos de qualidade a serem atendidos pela arquitetura.

Sub-atividade PD.2.2. Seleção das diretrizes arquiteturais de variabilidade baseada em DSLs

Como discutido no início deste capítulo, variabilidades mais complexas requerem uma descrição mais rica. Esta abordagem propõe o uso de linguagens específicas de domínio para formalizar o espaço de variação em áreas particulares do domínio (subdomínios), definindo os conceitos e introduzindo restrições e regras relacionadas à variabilidade neste subdomínio. DSLs também são utilizadas para guiar o desenvolvimento de transformações de software e geração de código, nas atividades de implementação.

Esta atividade cuida apenas da seleção de DSLs que descrevem o subdomínio relacionado ao módulo sendo refinado. Caso seja necessário desenvolver uma nova DSL, esta deve ser realizada posteriormente, na fase de implementação do domínio.

Sub-atividade PD.2.3. Seleção das diretrizes arquiteturais de integração entre subdomínios

É importante ressaltar que subdomínios quase nunca estão isolados entre si. Por exemplo, com relação ao domínio web de autoria de conteúdo mostrado na Figura 11, o subdomínio de navegação pode conter referências para o subdomínio de autoria (por exemplo, um formulário que aponta para um tipo específico de documento).

Assim, a arquitetura precisa estar preparada não só para a existência de múltiplos subdomínios e possivelmente múltiplas DSLs (WARMER; KLEPPE, 2006; HESSELLUND; CZARNECKI; WASOWSKI, 2007), mas também para a interação entre eles. Pode ser necessário, por exemplo, desenvolver um único metamodelo para os múltiplos subdomínios, e então desenvolver diferentes sintaxes concretas, de forma que estes subdomínios estejam integrados mas ainda assim possuir diferentes visões. Essa questão de integração de domínios é um assunto ainda não dominado.

Nesta atividade, estas interdependências entre subdomínios são explicitadas, pois podem ter impacto nas decisões de projeto. Para isso, cada elemento em cada subdomínio deve ser inspecionado, e cada elemento que depende de um elemento em um subdomínio diferente deve ser documentado.

As restrições de inclusão e exclusão mútua entre as *features* indicam algumas destas dependências, porém outras podem se tornar aparentes somente após a implementação estar avançada. Por este motivo, esta tese defende a idéia de que a identificação dos subdomínios deve acontecer de forma evolutiva e investigativa, desde a análise (Atividade AD.5). A cada iteração, os subdomínios devem evoluir, com o avanço da implementação, aumentando a confiança em sua validade e identificando novas relações entre os subdomínios. Na Seção 4.4 é apresentado um modelo de ciclo de vida que discute este aspecto interativo e iterativo da abordagem proposta.

Além das relações entre os subdomínios, o relacionamento entre cada subdomínio e as outras partes do domínio também precisam ser documentadas, de forma que é possível projetar uma arquitetura que acomode tanto artefatos gerados como não-gerados.

Atividade PD.3. Escolha de táticas e padrões arquiteturais

Papéis: projetista do domínio, especialista do domínio

Entradas: *PT.7. Diretrizes arquiteturais*

Saídas: *PT.8. Táticas e padrões arquiteturais*

Descrição: o objetivo desta atividade é selecionar ou definir táticas e padrões para satisfazer às diretrizes arquiteturais. O uso de padrões é considerado uma das mais técnicas mais úteis durante a fase de projeto de software. Através desta técnica, pode-se reaproveitar soluções recorrentes e já comprovadamente bem sucedidas, poupando-se esforço e riscos nesta atividade. Em termos de projeto arquitetural, são conhecidos como padrões, estilos (BASS; CLEMENTS; KAZMAN, 2003) ou abordagens (CLEMENTS; KAZMAN; KLEIN, 2004) arquiteturais. A diferença entre um padrão arquitetural e um padrão de projeto está na abrangência e impacto da solução. Enquanto padrões de projeto normalmente são utilizados para resolver problemas mais detalhados, padrões arquiteturais descrevem soluções mais abrangentes que envolvem todo o escopo de um domínio ou sistema. Além disso, padrões arquiteturais normalmente são escolhidos e utilizados conforme requisitos não-funcionais, como desempenho ou confiabilidade. Normalmente, inicia-se com padrões arquiteturais que definem a macro-estrutura do domínio, até chegar em padrões menores que resolvem problemas mais específicos de partes do domínio.

Além disso, um padrão arquitetural implementa uma ou mais táticas, que por sua vez tem o objetivo de alcançar um ou mais atributos de qualidade (BASS; CLEMENTS; KAZMAN, 2003). Uma tática descreve uma estratégia a ser utilizada para se alcançar um determinado atributo de qualidade. Assim, para cada diretriz arquitetural, seleciona-se uma ou mais táticas, e em seguida são selecionados os padrões para implementar estas táticas. Uma lista de táticas é apresentada por Bass, Clements e Kazman (2003), porém esta não inclui táticas para lidar com variabilidade em um domínio, nem com a integração entre subdomínios automatizados, como é o caso desta abordagem (Sub-atividade PD3.3).

Para implementar as táticas, existe uma grande quantidade de fontes de padrões e estilos arquiteturais. Vale destacar a série de cinco volumes conhecida como POSA (*Pattern-Oriented Software Architecture*) (BUSCHMANN et al., 1996; SCHMIDT et al., 2000; KIRCHER; JAIN, 2003; BUSCHMANN; HENNEY; SCHMIDT, 2007a, 2007b), que apresenta diversos padrões voltados para diferentes tipos de problemas arquiteturais. O catálogo clássico de padrões de projeto (GAMMA et al., 1995) também pode ser útil nesta atividade, além de outras fontes citadas por Bass, Clements e Kazman (2003).

Com relação ao problema da variabilidade, existem alguns trabalhos que propõem o uso de alguns padrões de projeto com este objetivo (ALMEIDA et al., 2007b; KEEPENCE; MANNION, 1999; LEE; KANG, 2004). Existem também diversos trabalhos que apresentam formas de se implementar variabilidade em uma linha de produtos, que podem ser adaptadas para o nível

de projeto arquitetural (ANASTASOPOULOS; GACEK, 2001; MUTHIG; PATZKE, 2003; RIBEIRO; MATOS; BORBA, 2008; SVANHBERG; GURP; BOSCH, 2002). Porém, estes não cobrem o aspecto de integração entre diferentes subdomínios. Para esta abordagem, foram selecionados alguns padrões que auxiliam na implementação das táticas específicas para variabilidade e integração entre subdomínios, apresentados mais adiante nesta seção.

A escolha das táticas e padrões a serem utilizados é guiada por dois fatores: o requisito em si, explicitado pela diretriz arquitetural, e os efeitos colaterais que o emprego de uma tática ou padrão provoca nas demais diretrizes (BASS; CLEMENTS; KAZMAN, 2003). Caso não seja possível encontrar alguma tática e/ou padrão que sirva para um cenário específico, pode-se modificar ou adaptar táticas e padrões existentes, ou mesmo criar novos especialmente para este domínio. Estes passam então a fazer parte do catálogo da organização e podem ser reaproveitados em projetos futuros.

Da mesma forma que as diretrizes arquiteturais, nesta abordagem são necessárias táticas e padrões para pelo menos três aspectos principais: variabilidade baseada em *features*, variabilidade baseada em DSLs e integração entre subdomínios. Alguns dos padrões identificados nesta tese são baseados nos trabalhos de Völter (2003), Völter e Bettin (2004), que são muito úteis para projetos de MDD, mas não são específicos para o projeto arquitetural ou para a engenharia de domínio, e portanto são acrescentadas discussões sobre como alguns desses padrões podem ser incorporados ao contexto de reutilização.

Sub-atividade PD.3.1. Padrões arquiteturais para variabilidade baseada em *features*

Existe uma série de padrões que podem ser utilizados para ajudar a tornar uma arquitetura de software específica de domínio preparada para os diferentes tipos de variabilidade baseada em *features*. Em particular, os padrões do GoF (GAMMA et al., 1995) podem facilitar a representação da variabilidade no projeto arquitetural, resolvendo alguns dos problemas relacionados à implementação das *features* (ALMEIDA et al., 2007b).

No caso do desenvolvimento orientado a modelos, é necessário considerar também como os geradores de código são integrados com cada padrão, já que partes do software são automaticamente geradas e precisam interagir com o restante do software.

O cenário é o seguinte: um modelo de *features* descreve os pontos comuns e variáveis. Um gerador de código usa como entrada uma seleção de *features/subfeatures* que faz parte da aplicação gerada, e precisa produzir o código correspondente. Para cada tipo de *feature*, um ou mais padrões do GoF (GAMMA et al., 1995) é utilizado.

***Features* alternativas:** estas são as *features* onde somente uma alternativa pode estar presente em uma aplicação.

Quando uma *feature* pode ser mapeada diretamente em uma única classe, o padrão *Abstract Factory* é indicado. Neste padrão, um elemento que realiza o papel de fábrica abstrata e um elemento que realiza o papel de produto abstrato representam um ponto de variação, uma *feature*. Fábricas e produtos concretos representam variantes alternativas. O gerador somente precisa gerar o código de instanciação da fábrica concreta correspondente, e o restante do código permanece independente.

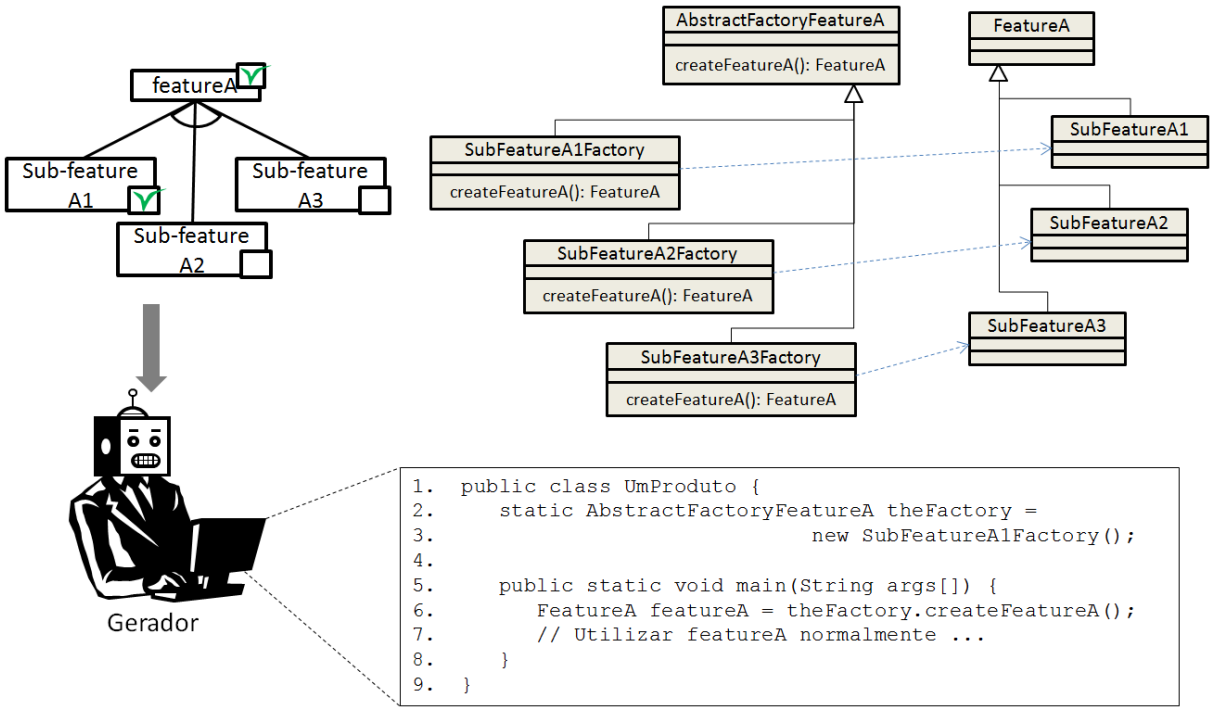


Figura 15: Uso do padrão *Abstract Factory* para *features* alternativas

A Figura 15 ilustra o uso deste padrão com um gerador de código para implementar *features* alternativas. Para cada ponto de variação, cria-se uma fábrica abstrata e um produto abstrato (*AbstractFactoryFeatureA* e *FeatureA*). Para cada variante alternativa, são criadas as fábricas e produtos concretos. O gerador, ao criar um produto, só precisa declarar a fábrica abstrata correspondente ao ponto de variação selecionado, neste caso a *featureA*, gerando as linhas 2 e 6, e instanciar a fábrica correspondente à alternativa selecionada (linha 3). O resto do código pode utilizar a *feature* normalmente.

O padrão *Prototype* pode ser utilizado com o mesmo propósito, nos casos onde se deseja evitar a criação de subclasses para os objetos construtores. Neste padrão, cada alternativa é implementada como uma classe diferente de um protótipo comum. O gerador de código é responsável por gerar código que instancia somente a alternativa selecionada.

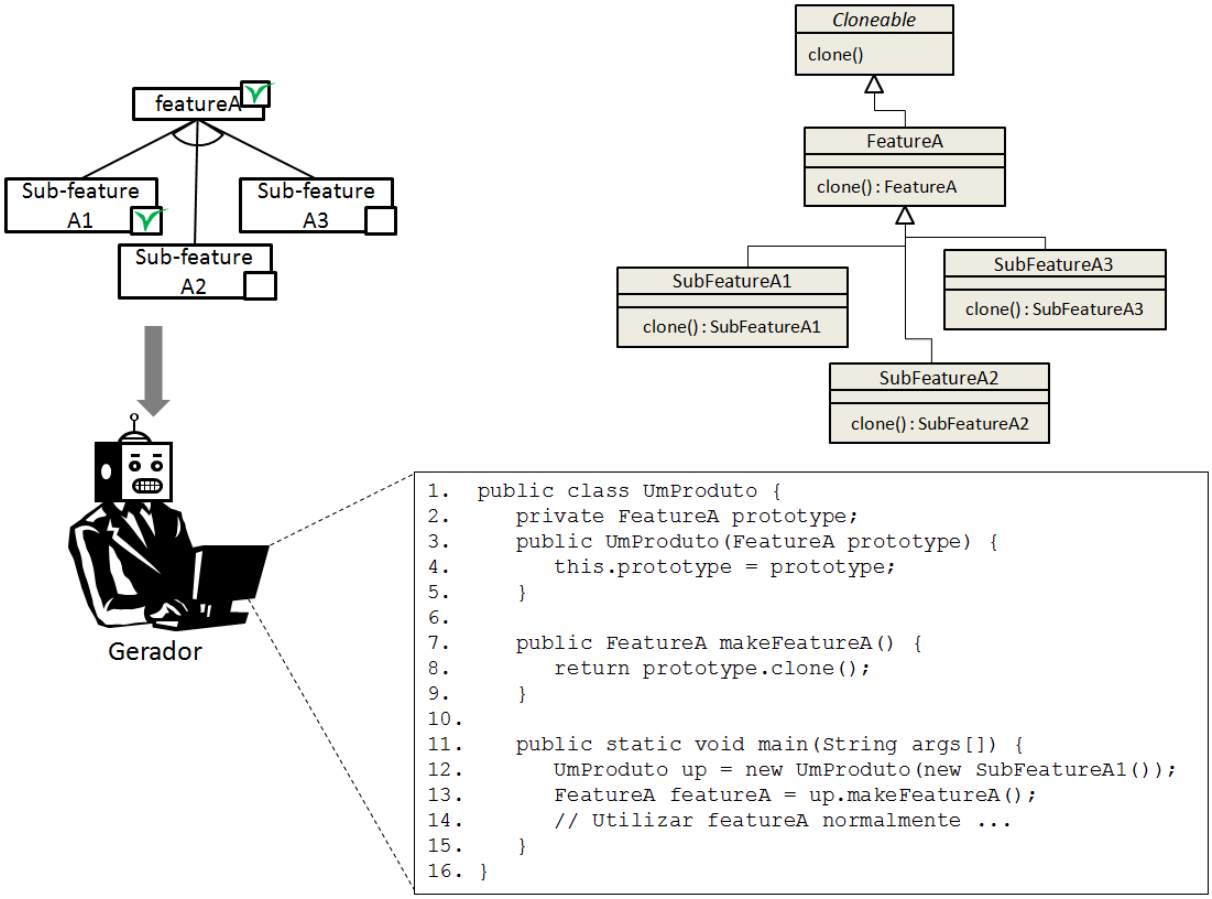


Figura 16: Uso do padrão *Prototype* para *features* alternativas

A Figura 16 ilustra o uso do padrão *Prototype* com um gerador de código para implementar *features* alternativas. Para cada ponto de variação, neste caso a *featureA*, cria-se um protótipo, que implementa uma interface (*Cloneable*) com um método para criar uma cópia de si mesmo (*clone()*). Cada variante alternativa (*Sub-features A1, A2 e A3*) é transformada em uma instância concreta do protótipo, mediante a implementação do método *clone()*, responsável por criar uma instância da variante.

O gerador de código, ao produzir o código do produto, só precisa criar as chamadas correspondentes à criação do ponto de variação e da alternativa selecionada. No exemplo da Figura 16, mediante a seleção do ponto de variação *featureA*, o gerador gera as linhas 2-9 e 13. A linha 12 contém o código que instancia a alternativa selecionada, no caso a variante *Sub-feature A1*.

Para o comportamento associado às *features*, uma solução é a utilização do *Template method*. A especificação do comportamento comum é definida em métodos abstratos de uma classe abstrata. O comportamento de cada alternativa é implementado em classes concretas, com implementações dos métodos correspondentes. A Figura 17 ilustra o uso deste padrão. O gerador apenas precisa gerar a criação do objeto concreto de acordo com a alternativa

selecionada (linha 3). Opcionalmente, a criação pode ser feita com o padrão *Abstract Factory*.

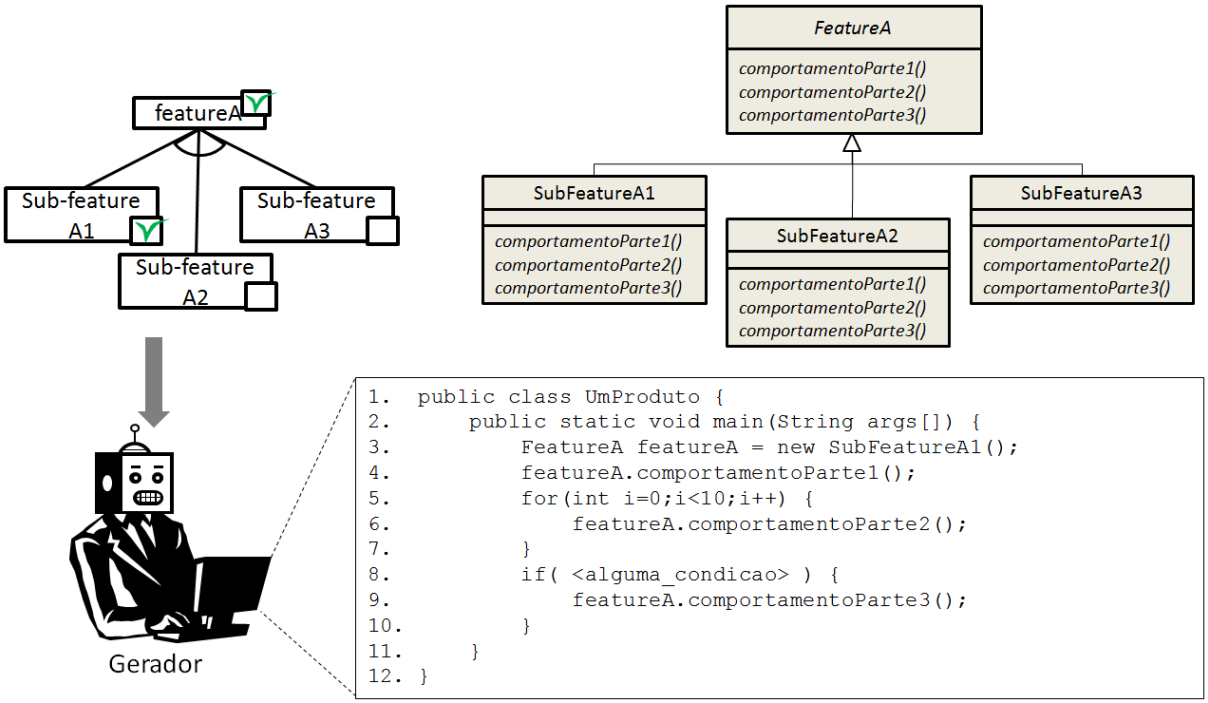


Figura 17: Uso do padrão *Template method* para *features* alternativas

Caso uma *feature* precise ser implementada por diferentes classes, sugere-se o uso do padrão *Facade* em conjunto com um dos três acima: *Abstract Factory*, *Prototype* ou *Template method*. O *Facade* permite a reunião de diversas classes em uma única "fachada". Neste caso, os métodos construtores (o construtor da classe, o método *clone()* ou o método *template*, nos padrões citados), precisam ser estendidos para instanciar todas as classes que implementam cada *feature*, com uma classe *facade* servindo para reuni-las em um único ponto.

Quando as *features* alternativas correspondem a comportamentos alternativos, que devem ser mapeados em um único método ao invés de toda uma classe, os padrões *Strategy* ou *Template method* podem ser utilizados. O padrão *Strategy* consiste na definição de uma interface, com um único método, que corresponde ao ponto de variação. Para cada alternativa deve-se providenciar uma implementação desta interface, onde o método contém o comportamento referente à alternativa selecionada. O *Template method* é similar, mas normalmente não exige uma interface dedicada a um único método. Em ambos os casos, o gerador produz as chamadas dos métodos correspondentes às alternativas selecionadas.

Or features: estas são similares às *features* alternativas, porém mais de uma *feature* pode estar presente em uma aplicação.

O padrão *Chain of Responsibility* pode ser utilizado quando as diferentes *features* introduzem funcionalidades complementares, que são executadas uma após a outra. Neste

padrão, cria-se uma classe abstrata para o ponto de variação, com um método principal e um método para adicionar comportamentos adicionais. Dentro do método principal, adiciona-se uma chamada para um comportamento abstrato, a ser implementado por cada instância concreta que corresponde às variantes.

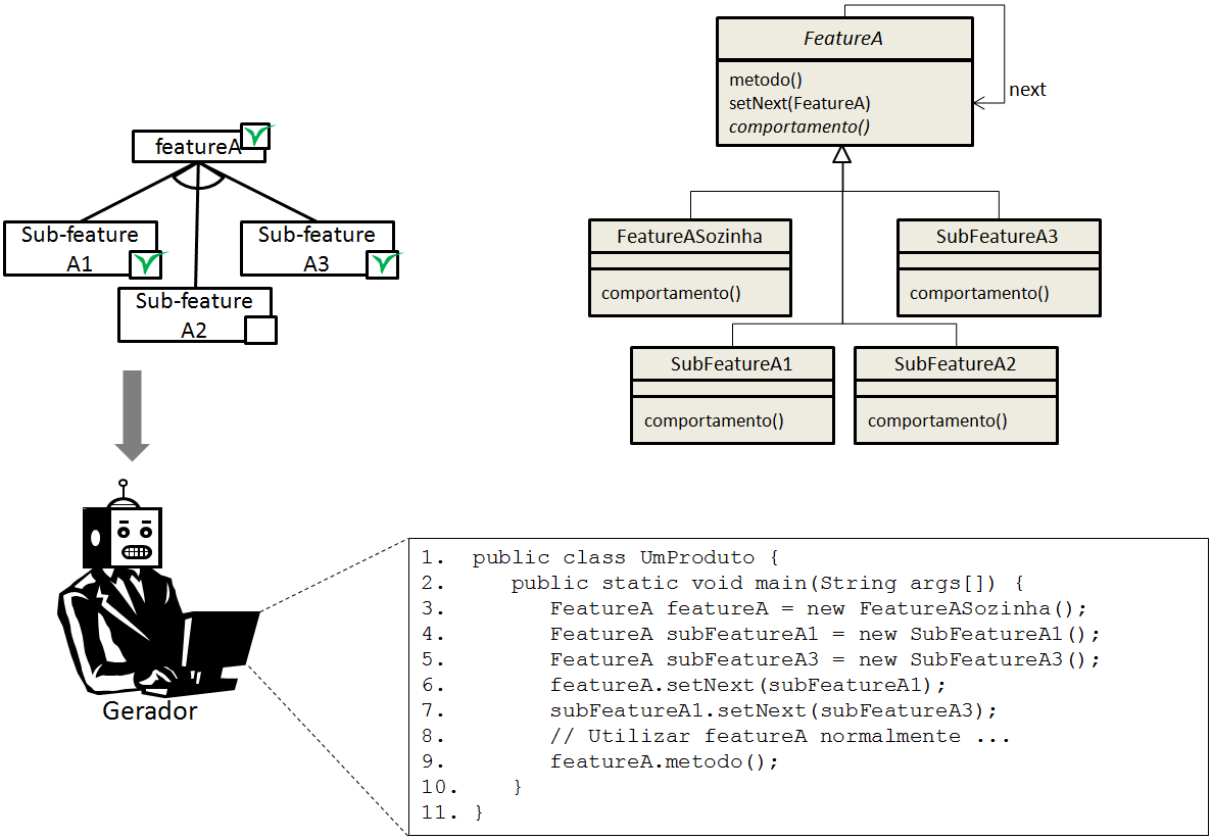


Figura 18: Uso do padrão *Chain of Responsibility* para *or features*

A Figura 18 ilustra o uso do padrão *Chain of Responsibility* na implementação de *or features*. Para cada ponto de variação, no caso *featureA*, cria-se uma classe abstrata, contendo um método principal (*metodo()*), a ser chamado no produto. Também é criado um método (*setNext()*) que permite "encadear" outras instâncias a esta. Para cada variante (*featureA* sozinha e *sub-features* A1, A2 e A3), cria-se uma subclasse que implementa o comportamento específico da variante. O gerador, para combinar as *features* selecionadas, cria uma instância da *feature* principal (linha 3), cria instâncias das *sub-features* selecionadas (linhas 4 e 5), e faz o encadeamento (linhas 6 e 7). Dessa forma, ao se chamar o método principal (linha 9), os comportamentos específicos de cada *sub-feature* são chamados.

O *Chain of Responsibility* permite a combinação de comportamentos sequenciais, ou seja, um é executado após o outro. Em interações mais complexas, onde a ordem de chamada dos comportamentos específicos não é sequencial, exigindo um código específico para isso, o padrão *Decorator* pode ser utilizado. Neste padrão, cria-se um *decorator* para cada variante.

Em cada *decorator* implementa-se uma versão dos métodos principais, considerando-se "como" esta variante modifica o comportamento normal. O gerador apenas precisa fazer a concatenação dos *decorators*, de forma a reproduzir as variantes selecionadas.

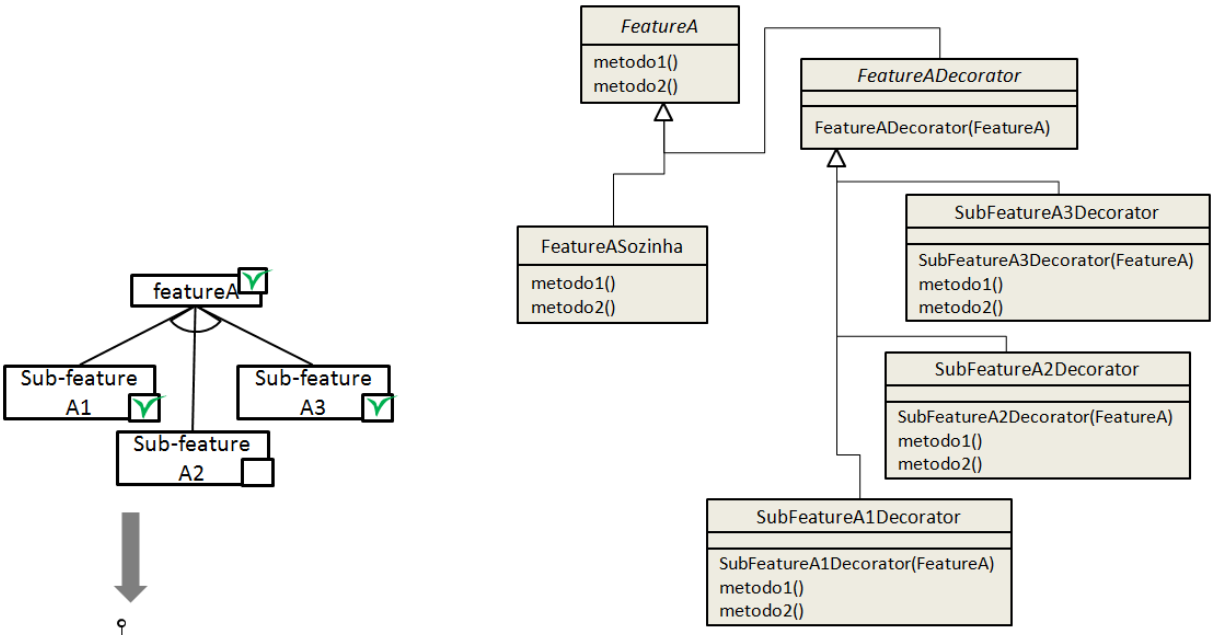


Figura 19: Uso do padrão *Decorator* para *or features*

A Figura 19 ilustra o uso do padrão *Decorator* para implementar *or features*. Para cada ponto de variação, no caso *featureA*, cria-se uma classe abstrata que contém métodos específicos desta *feature* (*metodo1()* e *metodo2()*). Cria-se também um *decorator* abstrato, com um construtor responsável por incluir o comportamento deste *decorator* a outro. Para cada variante (no caso, *featureA* sozinha, e as *sub-features* A1, A2 e A3), cria-se uma instância da classe abstrata principal (*FeatureASozinha*), e dos *decorators* (*SubFeatureA1Decorator*, *SubFeatureA2Decorator* e *SubFeatureA3Decorator*).

O gerador apenas precisa fazer a chamada que cria instâncias dos *decorators* que correspondem às variantes selecionadas (linhas 3, 4 e 5).

Da mesma forma que com as *features* alternativas, caso uma *or feature* precise ser implementada em várias classes, pode-se combinar o padrão *Facade* para reunir mais de uma classe em um único ponto.

***Features* opcionais:** para *features* opcionais, o mesmo conjunto de padrões utilizados para as *or features* podem ser utilizados, com a diferença de que neste caso não é necessário garantir que ao menos uma *feature* esteja presente na aplicação.

Para a implementação de variabilidade baseada em *features*, podem também ser utilizados dois padrões baseados em práticas bem conhecidas para a escrita de geradores de código (CZARNECKI et al., 2002).

O primeiro padrão é conhecido como a abordagem *visitante* (CZARNECKI; HELSEN, 2006; JOUAULT; BÉZIVIN; KURTEV, 2006; FEILKAS, 2006). Neste padrão, o modelo de entrada é percorrido e cada elemento é visitado. Para cada elemento, um *template* correspondente é chamado, de acordo com o tipo do elemento. No cenário de engenharia de domínio, este padrão é particularmente útil para diferentes tipos de *features* mandatórias e opcionais. A Figura 20 ilustra este padrão.

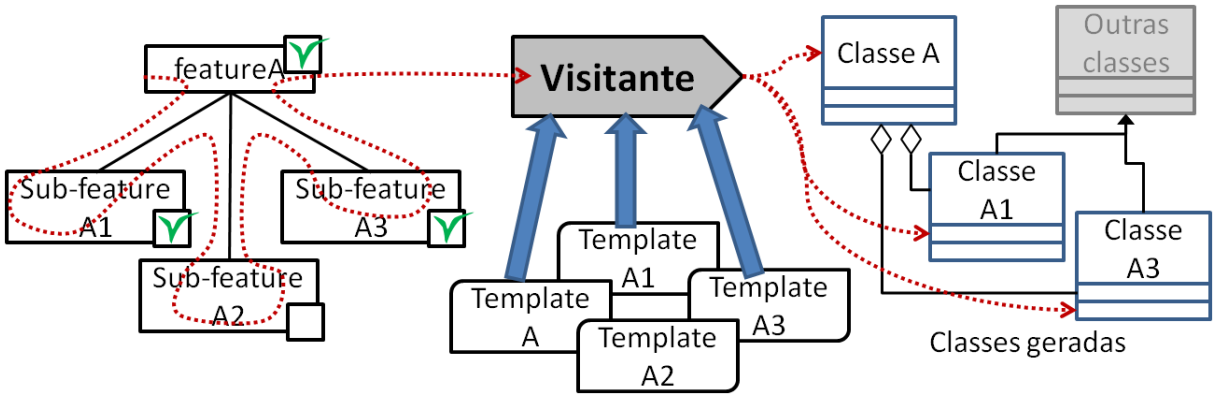


Figura 20: Padrão *visitante* sendo aplicado à implementação de variabilidade baseada em *features*

Neste exemplo, um visitante percorre o modelo de *features* e, para cada *feature* selecionada, chama o *template* correspondente, que gera código para ela. Normalmente, cada *template* produz uma única classe que implementa a *feature*, que se encaixa na arquitetura através de padrões de projeto como os descritos anteriormente nesta seção.

O padrão visitante é uma boa opção quando é possível encapsular a funcionalidade de uma *feature* em uma única classe. Caso não seja possível, a abordagem *template* (CZARNECKI; HELSEN, 2006) pode ser utilizada. Consiste em um único *template* que é o ponto de entrada, responsável por consultar os modelos e chamar outros *templates*. Esta abordagem difere da abordagem visitante no sentido em que a ordem e lógica por trás das chamadas dos *templates* é explicitamente programada pelo desenvolvedor, não dependendo de alguma política pré-estabelecida. Além disso, um *template* não necessariamente produz uma unidade distinta como uma classe. Um *template* pode introduzir apenas um único método, um pedaço de texto,

que pode ser inserido em outros artefatos. Também pode produzir hierarquias completas de classes.

Esta abordagem pode ser utilizada em diferentes tipos de variabilidade, por ser mais flexível, sendo particularmente útil para implementar *or features*, como ilustra a Figura 21.

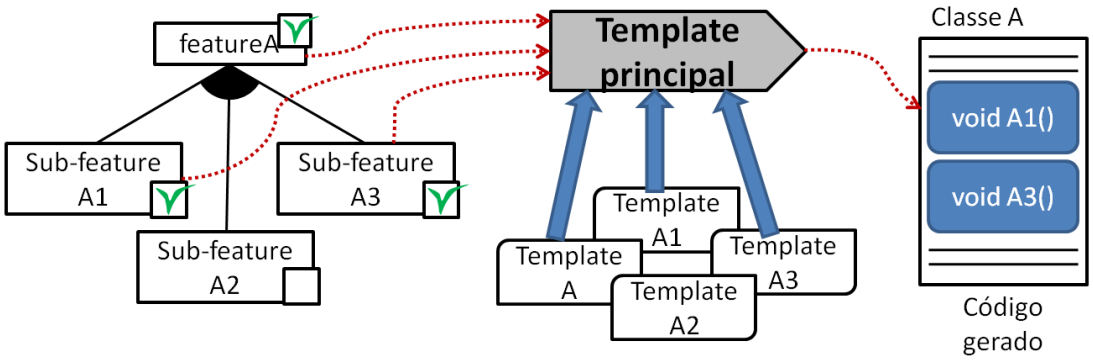


Figura 21: Abordagem *template* sendo aplicada à geração de código baseada em *features*

O *template* principal é responsável por analisar os modelos de *features* e decidir quais *templates* serão chamados, quando serão chamados, e em qual ordem. No exemplo da Figura 21, a *featureA* é selecionada, e é implementada por uma única classe, enquanto as *sub-features* A1 e A3 (que estão selecionadas), são implementadas como os métodos *A1()* e *A3()*.

Sub-atividade PD.3.2. Padrões arquiteturais para variabilidade baseada em DSLs

Este tipo de variabilidade é expresso em termos de uma DSL. O desenvolvedor especifica um programa/modelo que segue a sintaxe de uma DSL, e um gerador produz código-fonte automaticamente. A variabilidade em uma DSL pode ser virtualmente tudo que pode ser definido em um metamodelo: atributos verdadeiro/falso ou *strings*, listas tipadas e coleções, enumerações e outros conceitos da orientação a objetos podem ser utilizados para descrever o espaço de variabilidade. Para consultar estas estruturas em um gerador, construções comuns à maioria das linguagem de programação, como condições e laços, podem ser utilizadas.

Devido à grande variedade destes tipos de variabilidade, aqui não se propõe nenhum tipo de padrão associado a um tipo particular de variação (como na seção anterior). Ao invés disso, os padrões nesta categoria são focados em como as ferramentas baseadas em DSL e geradores de código podem ser integrados à arquitetura e aos geradores de código baseados em *features*. Apesar destes padrões não aparecerem na arquitetura da aplicação final, eles podem ter impacto no sucesso do domínio. Afinal, no MDD estes artefatos também fazem parte da arquitetura do domínio (VÖLTER; GROHER, 2007), e devem ser considerados durante o seu ciclo de vida.

Um primeiro padrão proposto denomina-se **camada fina de dados**, que facilita a

integração entre o gerador e a ferramenta de modelagem DSL. Normalmente, geradores de código consultam informações diretamente de uma ferramenta de DSL, que pode ser criada manualmente ou através de algum *framework* de construção de linguagens, como GME (*Generic Modeling Environment*), GMF (*Graphical Modeling Framework*) ou *openArchitectureWare* (Seção 2.2.2). Este padrão defende o uso de uma camada separada de dados, construída em uma tecnologia independente da ferramenta DSL, e que contém apenas as informações essenciais para o gerador, e nada mais.

Desta maneira, a informação necessária para o funcionamento do gerador é explicitada, facilitando a evolução independente do gerador e da ferramenta DSL. Também permite o desenvolvimento dos trabalhos de ambas as equipes: a que está trabalhando no gerador e outra equipe trabalhando na linguagem e suporte ferramental. Finalmente, este padrão libera o gerador de uma tecnologia de modelagem em particular, além de restringir as necessidades de aprendizado das particularidades das ferramentas de modelagem a uma única equipe. A equipe trabalhando com geração de código pode focar em suas próprias tarefas. A Figura 22 ilustra este padrão.

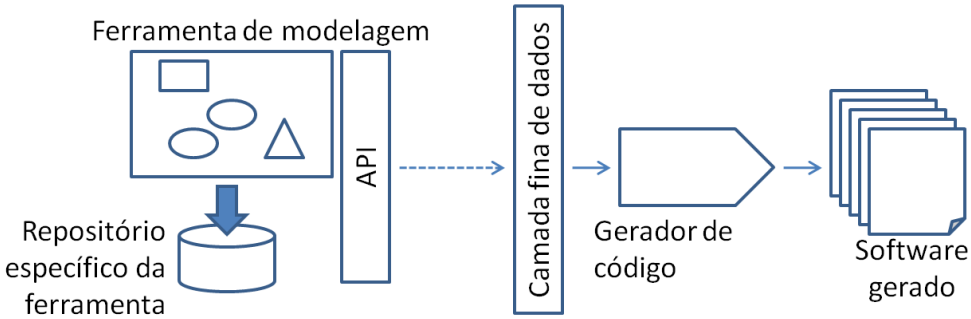


Figura 22: Padrão camada fina de dados

Um segundo padrão que pode ser utilizado, que deriva da camada fina de dados, é chamado **camada de dados das features**, e consiste numa especialização do padrão anterior. Normalmente, o modelo de *features* é o ponto central de informações para os geradores, mesmo aqueles exclusivamente dedicados à variabilidade baseada em DSLs. Neste padrão, que é uma instância do padrão *camada fina de dados*, propõe-se o uso de uma camada de dados que armazena todas as informações relacionadas às *features*. Esta camada de dados deve ser projetada para ser acessível a todos os geradores, permitindo que consultem informações das *features* enquanto geram código. Se existir uma ferramenta dedicada à atividade de modelagem de *features*, esta camada pode ser utilizada para fazer com que os geradores não dependam desta ferramenta em particular.

Sub-atividade PD.3.3. Padrões de integração entre subdomínios

Estes padrões buscam promover uma boa integração entre código gerado e não-gerado, particularmente nas áreas que envolvem os limites de um subdomínio. Os padrões descritos nesta seção dividem-se em quatro categorias, dependendo da dependência entre código gerado e não-gerado, conforme descrito a seguir.

Código gerado depende de código não-gerado: este é o tipo mais simples de interação, e consiste em fazer o gerador produzir código que usa código existente, não-gerado, como um *framework* ou biblioteca.

O padrão *Facade* (GAMMA et al., 1995) pode ser utilizado para simplificar a interação entre código gerado e não-gerado. Ao invés de gerar código que usa múltiplas classes e múltiplos métodos, uma única classe *Facade* agrupa todas as classes e métodos em um único ponto. Isto não somente torna as dependências mais explícitas, mas também promove alguma proteção contra mudanças no código não-gerado. Dependendo do grau das mudanças, pode não ser necessário modificar o gerador, o que é uma tarefa mais complexa e propensa a erros. Adaptações menores podem ser feitas diretamente na classe *Facade*.

Para proteger o gerador de mudanças maiores no código gerado, e algumas vezes simplificar o gerador, o padrão *Adapter* (GAMMA et al., 1995) pode ser utilizado, para coletar, filtrar e/ou preparar a informação necessária para o código gerado. Mudanças mais simples como no formato de dados ou na assinatura dos métodos não se propagam para o gerador.

Código não-gerado depende de código gerado: Isto acontece quando código não-gerado espera que um comportamento ou estrutura seja gerado.

É completamente possível e algumas vezes necessário esse tipo de padrão de integração, mas fazer com que um código não-gerado dependa diretamente de código gerado pode levar a problemas. O programa pode não compilar antes da geração completa de código. Objetos de exemplo podem ser manualmente criados temporariamente, mas em geral isto adiciona um nível a mais de dificuldade para testes/manutenção. Além disso, erros não previstos são mais prováveis de acontecer dependendo de como o código gerado varia, pois é difícil prever todas as possibilidades. Estes padrões visam reduzir estes problemas.

Na maioria dos casos, padrões como o *Template method* ou *Factory* (GAMMA et al., 1995) podem ser utilizados, de forma que o código não-gerado não precise saber detalhes sobre como as classes e métodos que o mesmo utiliza são implementados, se são gerados ou construídos manualmente. Porém, em algum momento, uma implementação concreta precisa ser providenciada, pois de outra forma o código não irá compilar e executar completamente.

Nesses casos, é possível remover a dependência entre código não-gerado e gerado por meio dos padrões de *Injeção de dependência* ou *Localizador de serviço* (FOWLER, 2004). Estes padrões removem as dependências do código (neste caso, do código não-gerado) e as colocam em agentes externos, responsáveis pela injeção das dependências através de configuração programática ou textual (normalmente XML). As dependências devem ainda ser definidas em algum lugar, mas uma vez que isto só ocorre em uma entidade separada, estas são explícitas, sendo também mais facilmente definidas, o que acaba melhorando o entendimento do código final. O código original pode ser compilado independentemente, facilitando testes e manutenção. Além disso, o código de configuração poderia ser gerado, de forma que até o processo de injeção seja automatizado. Por exemplo, é possível gerar arquivos de configuração para *frameworks* de injeção de dependência como o Spring[1].

Estes padrões assumem que há um elemento fixo entre os dois lados da dependência: uma interface, uma classe abstrata ou uma assinatura de método. Porém, há casos onde até mesmo as assinaturas dos métodos são geradas, como por exemplo um gerador que produz objetos de acesso a dados (*Data Access objects* - DAO), com métodos para realizar operações básicas de inserção, remoção e consulta. Cada DAO possui seus próprios conjuntos de métodos com diferentes assinaturas dependendo dos nomes e atributos das entidades. Nestes casos, técnicas de reflexão podem ser a única solução para remover dependências em tempo de compilação. Através da reflexão, é possível transformar todas as chamadas de métodos de código não-gerado para código gerado em chamadas reflexivas, de forma que o método sendo chamado é desconhecido pelo compilador.

Código gerado depende de código gerado: isto normalmente acontece quando há dois subdomínios que dependem um do outro.

Um problema que surge do relacionamento entre múltiplos subdomínios é como garantir a integridade deste relacionamento. Uma possibilidade é utilizar os nomes dos elementos como referências, isto é, o nome de uma referência em um modelo deve ser idêntico ao nome do elemento referenciado, em outro modelo. Apesar de não ser ideal, esta abordagem simplifica o processo de se implementar referências entre modelos. É efetivamente utilizada devido à sua praticidade, sendo encontrada em exemplos como *Apache OFBiz* e algumas *Microsoft DSL Software Factories* (HESSELLUND; CZARNECKI; WASOWSKI, 2007).

Outra opção são as pontes entre modelos, que consistem na criação de duplicatas de elementos, no metamodelo que contém a referência, que correspondem a elementos do metamodelo referenciado. No exemplo do domínio web de autoria de conteúdo,

[1] `http://www.springsource.org`

isto corresponde à criação, no metamodelo de navegação, de um elemento chamado *ReferenciaParaTipoDeDocumento*, que pode ser utilizado para estabelecer referências para o elemento *TipoDeDocumento* do metamodelo de autoria. Um verificador separado pode então ajudar a garantir que estas referências sejam válidas (WARMER; KLEPPE, 2006).

Código gerado precisa ser estendido: um exemplo típico é a geração de esqueletos de classes que precisam ser manualmente implementados após a geração.

Seja qual for a técnica sendo utilizada, uma recomendação importante é a de que a modificação manual do código gerado não deveria ser necessária (TOLVANEN, 2004; VÖLTER; BETTIN, 2004). Mesmo com técnicas de "*merging*" (mesclagem), como os blocos de usuário do JET (*Java Emitter Templates* (POPMA, 2004)), esta não é uma boa prática porque depende da existência de código que marca os locais onde o código manual começa e onde termina. Se este código for removido por alguma razão, o código manual pode ser perdido após a regeneração. Assim, técnicas de "*merging*" somente devem ser utilizadas quando estritamente necessárias, e de preferência com mecanismos que protegem o código manual de modificações acidentais (WARMER; KLEPPE, 2006).

As melhores táticas para evitar esta situação incluem a geração de classes abstratas ou interfaces, e a utilização de subclasses para implementar as partes faltantes. Técnicas como as receitas (*recipes*) do *openArchitectureWare*, que consistem na exibição de avisos sobre os passos a serem seguidos para completar o código, podem ajudar a garantir uma implementação manual correta.

Um padrão útil nestas situações é chamado *"merging" de geradores*. A tática por trás deste padrão é criar um modelo separado para a especificação das partes faltantes, e então combinar estes modelos utilizando um gerador específico. A Figura 23 ilustra a idéia.

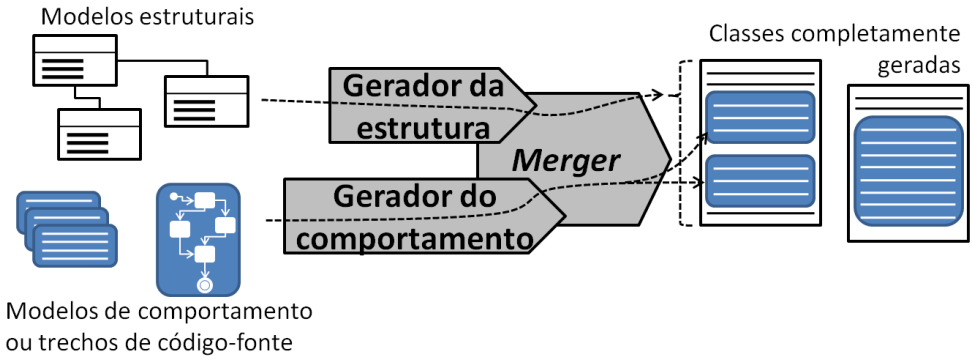


Figura 23: "*Merging*" de geradores envolvendo modelos estruturais e comportamentais

Neste exemplo, dois tipos de geradores – para modelos estruturais e de comportamento – são combinados para produzir código que não precisa ser manualmente completado. O

comportamento pode ser especificado por modelos específicos, como máquinas de estados, ou mesmo diretamente em código-fonte na linguagem de destino. Um gerador específico (*"merger"*) então traduz e/ou copia estes trechos para os locais designados das estruturas geradas.

Atividade PD.4. Aplicação dos padrões arquiteturais e refinamento dos módulos

Papéis: projetista do domínio, especialista do domínio

Entradas: *PT.6. Módulos a serem refinados*, *PT.7. Diretrizes arquiteturais* e *PT.8. Táticas e padrões arquiteturais*

Saídas: *PT.9.Inicial. Projeto do domínio*

Descrição: nesta atividade, os padrões são aplicados de acordo com as diretrizes selecionadas, e guiam o refinamento do domínio em módulos e sub-módulos. Por analogia, um padrão é como um *template*, onde seus elementos são papéis abstratos que precisam ser preenchidos por elementos concretos. A atividade anterior cuidou da definição deste *template*. Esta atividade cuida do preenchimento do *template*.

Como descrito na atividade PD.1, o refinamento ocorre em duas dimensões: de ponto comum para ponto variável e de módulo para sub-módulo. No caso de um refinamento na dimensão de **inclusão de ponto de variação**, definem-se quais novos elementos serão necessários para implementar este ponto de variação. O resultado é o surgimento de novos módulos que implementam o ponto de variação de acordo com táticas e padrões já testados e comprovados. No caso de um refinamento da dimensão de divisão **módulo/sub-módulo**, definem-se quais novos sub-módulos irão realizar os papéis do padrão.

Este refinamento é guiado pelo padrão sendo aplicado. Por exemplo, caso o padrão *Abstract Factory* seja aplicado em um módulo, irão surgir novos módulos que correspondem às diferentes fábricas e produtos concretos. O mesmo ocorre no nível arquitetural. Caso seja aplicado o padrão em camadas, o refinamento produz novos módulos, um para cada camada, por exemplo. O resultado é uma decomposição plausível, guiada por um padrão que tem por objetivo atender às necessidades arquiteturais específicas daquele módulo (diretrizes) (BASS; CLEMENTS; KAZMAN, 2003).

Nesta atividade, são também descritas as interfaces dos módulos recém criados. Além das informações requeridas/providas para que cada módulo seja capaz de executar sua responsabilidade, são descritos também os requisitos e funções que devem ser atendidos por

aquele módulo específico, considerando-se a divisão que foi realizada.

Por exemplo, para satisfazer a um determinado cenário de variabilidade onde uma *feature* pode ou não estar presente (*feature* opcional), pode-se criar um módulo que aceita como parâmetro uma variável *booleana* que indica se a *feature* está ou não presente. Assim, a definição da interface deste módulo deve conter uma descrição desta variável, do seu funcionamento, e dos requisitos que levaram à sua criação.

Todos os requisitos e funções associados com o módulo pai devem possuir um módulo correspondente que assuma a responsabilidade. Estas responsabilidades devem estar claramente descritas e indicadas nas interfaces.

Atividade PD.5. Avaliação arquitetural

Papéis: Projetista do domínio, especialista do domínio, demais *stakeholders*

Entradas: *PT.7. Diretrizes arquiteturais* e *PT.8.Inicial. Projeto do domínio*

Saídas: *PT.9.Avaliado. Projeto do domínio*

Descrição: a abordagem segue o raciocínio do método PuLSE-DSSA (DEBAUD; FLEGE; KNAUBER, 1998), no sentido em que o projeto arquitetural pode produzir múltiplas arquiteturas, cada uma oferecendo uma alternativa para se atender às diferentes diretrizes. Uma atividade é então responsável por avaliar as alternativas e selecionar qual delas segue adiante no processo.

A avaliação arquitetural deve ser iniciada quando a equipe de desenvolvimento começar a tomar decisões que dependem da arquitetura, e o custo de se desfazer tais decisões é maior do que o custo de se realizar a avaliação (CLEMENTS; KAZMAN; KLEIN, 2004). Nesta abordagem, estas decisões são referentes à automação dos subdomínios, utilizando ferramentas de modelagem e transformadores, que depende desta estrutura em comum para possibilitar a reutilização.

Assim, esta atividade busca avaliar as alternativas de arquiteturas projetadas anteriormente. O objetivo é selecionar aquela que melhor atende aos requisitos para o domínio, com foco na variabilidade. Este pode parecer um passo desnecessário, uma vez que na atividade anterior o projeto arquitetural foi realizado com base em um conjunto de diretrizes que representam os mesmos requisitos. A diferença, no entanto, é que agora serão incluídos os pontos de vista dos outros interessados no domínio, para serem confrontados com os requisitos iniciais, o que potencialmente irá revelar alguns aspectos que foram ignorados pelo projetista.

Este confronto irá não somente facilitar a seleção de uma arquitetura adequada por meio

de consenso entre todos os envolvidos, como também irá resultar em possíveis sugestões de alteração na arquitetura, visando atender os cenários não originalmente previstos. Assim, esta atividade também pode ser vista como uma atividade de melhoria e evolução da arquitetura.

Esta atividade é inspirada no SAAM (*Software Architecture Analysis Method* ou Método de Avaliação de Arquitetura de Software) (CLEMENTS; KAZMAN; KLEIN, 2004), com 3 passos:

1. **Desenvolver cenários**: cenários, conforme discutido anteriormente, capturam os principais requisitos e funcionalidades que devem ser atingidos pela arquitetura. Alguns já foram definidos anteriormente, incluindo os casos de uso, de mudança e os principais requisitos de qualidade. Estes foram elaborados principalmente pelo projetista e especialista do domínio. Aqui, o foco é tentar reduzir o número de cenários para aqueles que possuem impacto na arquitetura, e tentar incluir o ponto de vista de outros *stakeholders*, tais como usuários finais, gerentes, testadores, etc. Pode-se utilizar sessões de *brainstorming* para capturar o máximo de informação possível;

2. **Avaliar cenários individualmente**: neste passo, a exemplo do método SAAM citeClements:2004:book:evaluating, cada cenário é avaliado de forma individual, por meio de *workshops* mediados pelo projetista, onde se discute como o cenário está ou não sendo atendido pelas diferentes arquiteturas. Pode-se também desenvolver protótipos (DEBAUD; FLEGE; KNAUBER, 1998) com funcionalidades simuladas que refletem o cenário sendo avaliado. Para cada cenário o projetista descreve como a arquitetura oferece o suporte, ou descreve as mudanças necessárias; e

3. **Avaliar interação entre os cenários**: uma vez que se tenha as descrições de mudanças referentes ao suporte aos cenários, este passo tem como objetivo destacar quais cenários interagem entre si, isto é, exigem mudanças no mesmo local ou grupo de módulos/componentes. Áreas onde existe grande interação entre cenários podem significar pontos onde o projetista falhou em corretamente implementar a separação de interesses, e portanto pode ser necessária maior atenção nesta área, ou mesmo uma nova passagem pelas atividades de projeto (CLEMENTS; KAZMAN; KLEIN, 2004).

O resultado destes passos é uma lista que descreve como cada arquitetura atende aos requisitos conforme expressos pelos *stakeholders*. Para cada arquitetura, pode-se calcular:

- **Número de cenários diretamente suportados:** para cada arquitetura, conta-se o número de cenários que foram avaliados como possuindo suporte direto da arquitetura, sem necessidade de alterações. Este número deve considerar o peso de cada cenário, conforme estipulado no passo 1 descrito anteriormente;

- **Número de cenários indiretamente suportados:** para cada arquitetura, conta-se o número de cenários que requerem alguma alteração para passarem a possuir o suporte adequado pela arquitetura. Este número deve considerar o peso de cada cenário, conforme estipulado no passo 1 descrito anteriormente; e
- **Quantidade de mudanças:** para cada arquitetura, estima-se a quantidade de mudanças necessárias para que a mesma passe a oferecer suporte a todos os cenários indiretos. Esta quantidade pode ser medida em termos de número de módulos/componentes modificados ou, caso já existam mais detalhes sobre a arquitetura, até mesmo o número de classes envolvidas nas mudanças.

De posse destes números, é possível determinar qual das alternativas oferece o melhor suporte global para os requisitos. A alternativa que apresentar o melhor suporte aos cenários, e menor quantidade de mudanças, será selecionada para seguir adiante. Uma vez selecionada a arquitetura, retorna-se às atividades PD.2 e PD.3, buscando novas táticas/padrões arquiteturais para implementar as mudanças sugeridas nesta avaliação. Este ciclo se repete até não serem necessárias mais mudanças para atender aos cenários.

Após esta atividade, tem-se a arquitetura projetada e avaliada com base nas informações disponíveis até o momento. No entanto, conforme já mencionado anteriormente, este é um processo iterativo, e a arquitetura pode vir a sofrer alterações posteriormente. Na realidade, o que acontece é uma iteração em duas vias: a arquitetura influencia a implementação das DSLs e transformadores, que por sua vez podem influenciar a arquitetura, resultando em mudanças que melhor acomodem a presença destes novos artefatos.

6.3 Considerações finais

Neste capítulo foi apresentada a fase de projeto de domínio, com atividades para promover a reutilização através do MDD. As principais contribuições deste capítulo são as atividades para definição das diretrizes e padrões arquiteturais específicos para o uso do MDD na reutilização de software: variabilidade baseada em *features*, variabilidade baseada em DSLs e integração entre subdomínios. Também apresenta uma atividade de avaliação arquitetural visando melhorar o resultado do projeto antes da implementação ter iniciado.

O Quadro 10 resume as atividades do projeto de domínio.

O Quadro 11 descreve os produtos de trabalho da fase da projeto de domínio.

O produto do projeto de domínio é a definição da arquitetura e seus componentes. Nesta

Atividades e sub-atividades	**Entradas**	**Saídas**
PD.1. Escolha dos módulos a serem refinados	PT.3.Validado. Modelagem do domínio PT.4.Validado. Candidatos a subdomínio PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios	PT.6. Módulos a serem refinados
PD.2. Seleção das diretrizes arquiteturais PD.2.1. Seleção das diretrizes arquiteturais de variabilidade baseada em *features* PD.2.2. Seleção das diretrizes arquiteturais de variabilidade baseada em DSLs PD.2.3. Seleção das diretrizes arquiteturais de integração entre subdomínios	PT.6. Módulos a serem refinados	PT.7. Diretrizes arquiteturais
PD.3. Escolha de táticas e padrões arquiteturais PD.3.1. Padrões arquiteturais para variabilidade baseada em *features* PD.3.2. Padrões arquiteturais para variabilidade baseada em DSLs PD.3.3. Padrões de integração entre subdomínios	PT.7. Diretrizes arquiteturais	PT.8. Táticas e padrões arquiteturais
PD.4. Aplicação dos padrões arquiteturais e refinamento dos módulos	PT.6. Módulos a serem refinados PT.7. Diretrizes arquiteturais PT.8. Táticas e padrões arquiteturais	PT.9.Inicial. Projeto do domínio
PD.5. Avaliação arquitetural	PT.7. Diretrizes arquiteturais PT.9.Inicial. Projeto do domínio	PT.9.Avaliado. Projeto do domínio

Quadro 10: Resumo das atividades para projeto de domínio orientado a modelos

abordagem, além de componentes de software tradicionais, a arquitetura comporta a existência de artefatos como DSLs, transformações e geradores de código.

Produto de trabalho	**Descrição**	**Estado**
PT.6. Módulos a serem refinados	Consiste na definição dos módulos a serem refinados em uma determinada iteração do projeto do domínio. Iniciando com o domínio todo, a cada etapa um ou mais módulos são refinados e produzem uma nova versão da arquitetura do domínio	Nenhum
PT.7. Diretrizes arquiteturais	Conjunto de requisitos importantes para a definição da arquitetura. Consiste em objetivos de negócio, casos de uso, cenários e linguagens que descrevem a variabilidade, além de uma lista de dependências entre os subdomínios	Nenhum
PT.8. Táticas e padrões arquiteturais	Descrevem soluções para problemas comuns no projeto arquitetural orientado a modelos	Nenhum
PT.9. Projeto do domínio	Resultado da fase de projeto, este produto de trabalho engloba a definição da arquitetura, com os módulos e sub-módulos, e a especificação das interfaces, considerando a existência de componentes tradicionais, e artefatos do MDD, como DSLs, ferramentas, transformações e geradores de código	1. *Inicial*: versão inicial do documento, gerada somente pelo projetista com o auxílio do especialista. Pode conter inconsistências ou não cumprir cenários que sejam importantes a algum outro *stakeholder* 2. *Avaliado*: projeto avaliado por uma atividade específica, que considera o ponto de vista de diversos *stakeholders*

Quadro 11: Descrição dos produtos de trabalho da fase de projeto de domínio

7 *Implementação de domínio orientada a modelos*

A análise de domínio e projeto arquitetural, realizados em fase anterior, lidam com questões como qual a melhor maneira de dividir o domínio em sub-sistemas ou como os módulos devem interagir entre si de forma a maximizar o desempenho na execução de determinada tarefa crítica. Porém, questões de mais baixo nível ainda permanecem não-resolvidas, tais como: qual tecnologia de comunicação deve ser utilizada em um domínio distribuído? Como será o acesso à base de dados? Como a internacionalização será alcançada? Qual algoritmo de busca deve ser utilizado? Estas são o foco da etapa de implementação do domínio, que nesta abordagem também engloba o refinamento do projeto de alto nível em um projeto detalhado.

Entre as questões a serem respondidas, destaca-se o suporte à variabilidade. Como os componentes do domínio irão dar suporte aos diferentes pontos de variação? Esta questão começou a ser respondida durante o projeto, e agora é necessário tomar as decisões finais quanto ao projeto detalhado. Além disso, sendo esta uma abordagem orientada a modelos, é necessário também implementar os artefatos específicos do MDD. Assim, a implementação deve também produzir linguagens específicas de domínio, transformações e geradores de código.

Nesta tese, entende-se por gerador de código qualquer artefato capaz de produzir automaticamente, com base em uma especificação de entrada, um trecho de código como saída. Este trecho de código pode ser qualquer tipo de elemento textual, mas normalmente corresponde a um programa escrito em uma linguagem de programação. Segundo esta definição, um processador automático de *templates*, como o JET, é um gerador de código. Mas um arquivo contendo um *template* também é considerado um gerador de código, uma vez que este possui instruções específicas que influenciam diretamente no código gerado.

Enquanto as fases de análise (ARANGO, 1999; PRIETO-DÍAZ, 1990; KANG et al., 1990; FRAKES; PRIETO-DÍAZ; FOX, 1998; BAYER; MUTHIG; WIDEN, 1999; KIM; YANG; PARK, 2003; MEI; ZHANG; GU, 2003; MOON; YEOM, 2005; ALMEIDA et al., 2006, 2005) e projeto (ALMEIDA et al., 2005; BOSCH, 2000; BASS; CLEMENTS; KAZMAN, 2003; TRACZ, 1995) de

domínio para reutilização são amplamente discutidas pela comunidade científica, a área de implementação apresenta algumas lacunas que precisam ser preenchidas (ALMEIDA et al., 2008; ANASTASOPOULOS; GACEK, 2001). Uma das razões é que a engenharia de domínio tem suas raízes na comunidade de reutilização de software, que favorece uma abordagem *top-down* (PATZKE; MUTHIG, 2002), dando mais ênfase à análise e projeto.

Outra razão é o fato de que, para definir um processo genérico, deve ser possível generalizar detalhes específicos de plataforma e linguagem, de forma que o processo possa ser utilizado independentemente da tecnologia de implementação. Ao mesmo tempo, a implementação é extremamente dependente da tecnologia (ALMEIDA et al., 2008), fazendo com que esta generalização seja uma luta entre forças opostas. Como resultado, observa-se uma falta de orientação com relação às atividades para implementação de um domínio para reutilização (PATZKE; MUTHIG, 2002).

Há inúmeras técnicas para se implementar domínios reutilizáveis, como aquelas descritas na Seção 3.2.1. Estas podem ser divididas em três dimensões: gerenciamento de configuração, tecnologias de componente e as características generativas das linguagens de programação (MUTHIG et al., 2002). Cada dimensão tem sua maneira particular para lidar com a variabilidade, que é o desafio mais notável na implementação de um domínio reutilizável. A dimensão de gerenciamento de configuração, por exemplo, gerencia as variantes alternativas de um mesmo artefato em um mesmo ponto no tempo (MUTHIG; PATZKE, 2004).

A dimensão de tecnologia de componentes baseia-se na idéia de componentes de software. Um componente é, por definição, uma unidade de composição (MUTHIG et al., 2002), e portanto esta dimensão lida com a variabilidade através da composição das variantes requeridas na forma de componentes (KETTEMANN; MUTHIG; ANASTASOPOLOUS, 2003; MUTHIG; PATZKE, 2004).

A tecnologia generativa (CZARNECKI; EISENECKER, 2000), foco desta tese, pode introduzir um controle mais poderoso sobre a variabilidade no domínio. Enquanto variações dentro de um componente tradicional limitam-se a estruturas fixas e interfaces previamente projetadas, a tecnologia generativa possibilita variação em menor granularidade, mesmo dentro de componentes. Fragmentos de código variante podem ser sistematicamente arranjados para formar a implementação de um componente (MUTHIG; PATZKE, 2004).

7.1 Objetivos da implementação do domínio

O objetivo desta fase é implementar o domínio, ou seja, implementar componentes, DSLs, transformações e geradores de código, seguindo o projeto definido na fase anterior.

Além disso, existe uma série de critérios que precisam ser atendidos pela implementação de um domínio, dos quais se destacam aqueles que resultam da necessidade de um bom gerenciamento da variabilidade (ANASTASOPOULOS; GACEK, 2001; MATOS JR, 2008; MUTHIG; PATZKE, 2003, 2004):

1. **Minimizar duplicação de código:** deve-se provocar pouca ou nenhuma duplicação de código ao implementar a variabilidade;

2. **Esforço de reutilização:** a implementação deve permitir a reutilização de forma simples e com pouco esforço, seguindo o princípio de que se for mais trabalhoso reutilizar do que construir, a reutilização não irá acontecer;

3. **Separação de interesses:** separação entre variantes e código comum, de forma que mudanças podem ser feitas separadamente em ambos os códigos;

4. **Desempenho:** a implementação deve proporcionar desempenho do produto final de acordo com os requisitos.

5. **Diferentes tipos de código-fonte**: os mecanismos normalmente citados na literatura são fortemente atrelados a uma linguagem de programação, e normalmente utilizam conceitos de orientação a objetos. Porém, padrões de projeto, herança, polimorfismo e orientação a aspectos, entre outros exemplos, fazem pouco sentido em artefatos como páginas HTML, *scripts* de criação de banco de dados, arquivos de configuração, *stored procedures*, entre outros tipos de artefatos de implementação que não são baseados em uma linguagem de programação. Obviamente, ainda é possível algum tipo de controle como, por exemplo, a extensão de uma página HTML com *scriptlets* que implementam a inclusão condicional de partes de texto correspondentes a variantes específicas. Porém, esta opção é mais indicada para variações em tempo de execução, não sendo adequada para as variações em tempo de compilação, que é o foco desta tese. Como resultado, nestes casos faz-se necessário o gerenciamento manual da variabilidade; e

6. **Variabilidades complexas**: conforme discutido no Capítulo 6, podem existir alguns casos de variabilidades mais complexas do que a modelagem de *features* é capaz de representar. Normalmente, são pontos de variação abertos, nas quais não é possível determinar um limite para a presença ou ausência de *features*, a sua implementação requer um esforço criativo composicional manual durante o desenvolvimento das aplicações. Apesar de ser possível facilitar este esforço através de padrões como *factories* (GAMMA et al., 1995) ou outras técnicas similares, o mesmo não pode ser completamente automatizado utilizando somente tais mecanismos.

A fase de implementação segue a mesma estrutura do método utilizado por Muthig e Patzke (2004). Inicialmente, são desenvolvidas as comunalidades, ou as *features* comuns a todos os produtos do domínio. Em seguida, as variabilidades são introduzidas, uma a uma, de forma que ao final tem-se uma implementação que cubra todas as possibilidades. Todo o processo é baseado em desenvolvimento através de exemplos (WIMMER et al., 2007), o que facilita a tarefa do desenvolvedor.

7.2 Atividades da implementação do domínio

Inicialmente, cada subdomínio identificado durante a análise de domínio é caracterizado em termos de sua variabilidade (Atividade ID.1). Em seguida são definidas as DSLs e o suporte ferramental (ferramentas de modelagem e editores específicos de domínio) (Atividade ID.2), por meio de uma abordagem *top-down* que utiliza os modelos de *features* para definir as DSLs para cada subdomínio. A seguir as DSLs são refinadas, e as transformações são construídas (Atividade ID.3), utilizando uma abordagem *bottom-up* e uma implementação de referência como ponto de partida. Esta implementação de referência é então transformada em um *framework* de domínio (Atividade ID.4), contendo classes e componentes reutilizáveis que dão suporte para algumas das variabilidades. Finalmente, os artefatos construídos são documentados (Atividade ID.5) de forma a facilitar seu entendimento, manutenção e reutilização.

A seguir estas atividades são descritas de forma detalhada.

Atividade ID.1. Caracterização da variabilidade dos subdomínios

Papéis: implementador do domínio, Especialista do domínio

Entradas: *PT.3.Validado. Modelagem do domínio*, *PT.4.Validado. Candidatos a subdomínio*, *PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios*, *PT.9.Avaliado. Projeto do domínio*

Saídas: *PT.10. subdomínios caracterizados*

Descrição: um domínio pode ser dividido em vários subdomínios, cada um com um potencial de automação. Durante a análise, esta divisão se inicia, com a identificação de candidatos a subdomínio (Atividade AD.3). Durante o projeto, são identificadas as diretrizes arquiteturais que apresentam mais detalhes sobre o tipo de variabilidade de cada subdomínio (Atividade PD.2). Nesta atividade, cada subdomínio é efetivamente caracterizado com relação à

sua variabilidade, visando dar subsídios à implementação do suporte em termos de modelagem, transformações e geradores de código para sua automação.

Conforme discutido no Capítulo 6, existem diferentes tipos de variabilidade, que são caracterizados de acordo com um espectro entre configuração de rotina e construção criativa. Nesta atividade, cada subdomínio é analisado e inserido em um determinado local deste espectro.

A Figura 24 ilustra esta estratégia. Para todo o domínio, uma ferramenta de *features* é normalmente utilizada, junto com uma ferramenta de configuração automática. Alguns subdomínios podem exigir uma solução baseada em uma DSL completa, incluindo uma ferramenta de modelagem e geradores dedicados. Em outros, um simples *wizard* pode ser suficiente.

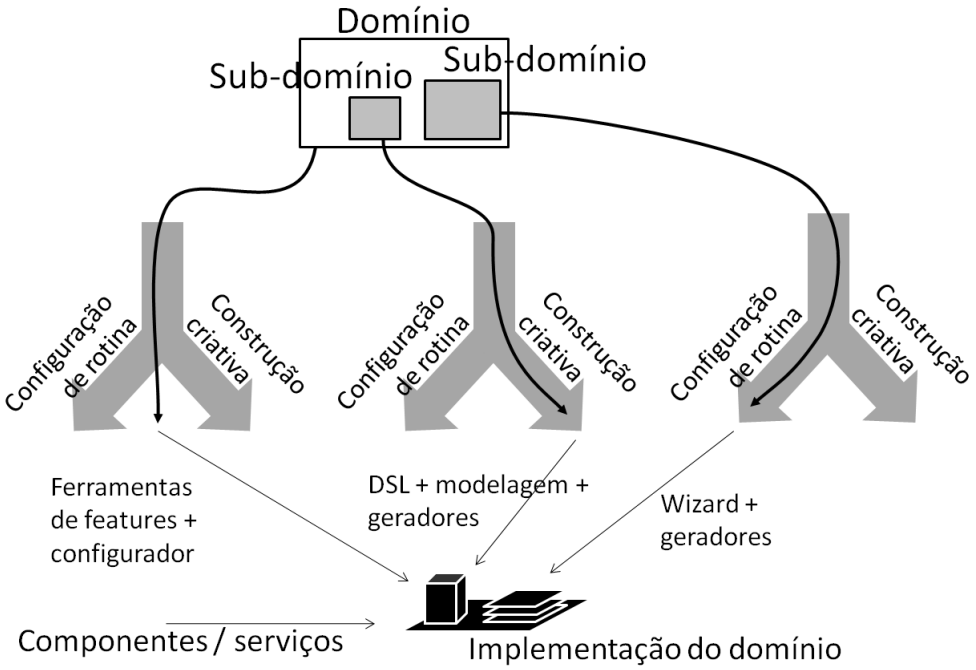


Figura 24: Estratégia de caracterização de subdomínios

Para inserir cada subdomínio em algum lugar do espectro de variabilidade, o papel do especialista do domínio é muito importante, mas as seguintes diretrizes podem ser úteis:

D_1. **Procurar por configurações de *features* que não mudam entre as aplicações:** se uma *feature* representa um ponto de variação, sua configuração deve mudar de alguma forma quando as diferentes aplicações variam com relação a este ponto. Por exemplo, se uma aplicação web provê busca avançada, e uma segunda aplicação provê busca simples, as configurações das *features* para estas aplicações serão diferentes. Isto indica que a variabilidade pode ser representada como *features*. Porém, se duas aplicações diferem em algum ponto, mas as configurações das *features* que descrevem aquele ponto são as mesmas, isto pode indicar que

há alguma variabilidade que não pode ser representada como *features*, e talvez seja necessária uma DSL. Por exemplo, considere duas aplicações web com suporte para busca avançada. Na primeira, é possível buscar por conteúdo de usuário pelo autor, data e palavras-chave. Na segunda, é possível buscar pelo autor, tipo do conteúdo, palavras-chave e outros metadados específicos desta aplicação. Neste exemplo, ambas as configurações irão apresentar a *feature* "Busca avançada", apesar de possuírem detalhes distintos.

D_2. **Analisar as *features* de tecnologia do domínio:** *features* de tecnologia do domínio representam as maneiras de se implementar os serviços ou operações do domínio (LEE; KANG; LEE, 2002). Eles são alguns dos "blocos de construção" sobre os quais a implementação é realizada, e normalmente é necessário um processo de construção criativa para combiná-los de forma a satisfazer os requisitos da aplicação. Portanto, há uma chance de que a variabilidade associada seja aberta. Por exemplo, as seis *features* de tecnologias de domínio localizadas no lado inferior da Figura 25 não podem ser meramente selecionadas ou de-selecionadas. Ao contrário, elas precisam ser combinadas de forma criativa ao se configurar um produto.

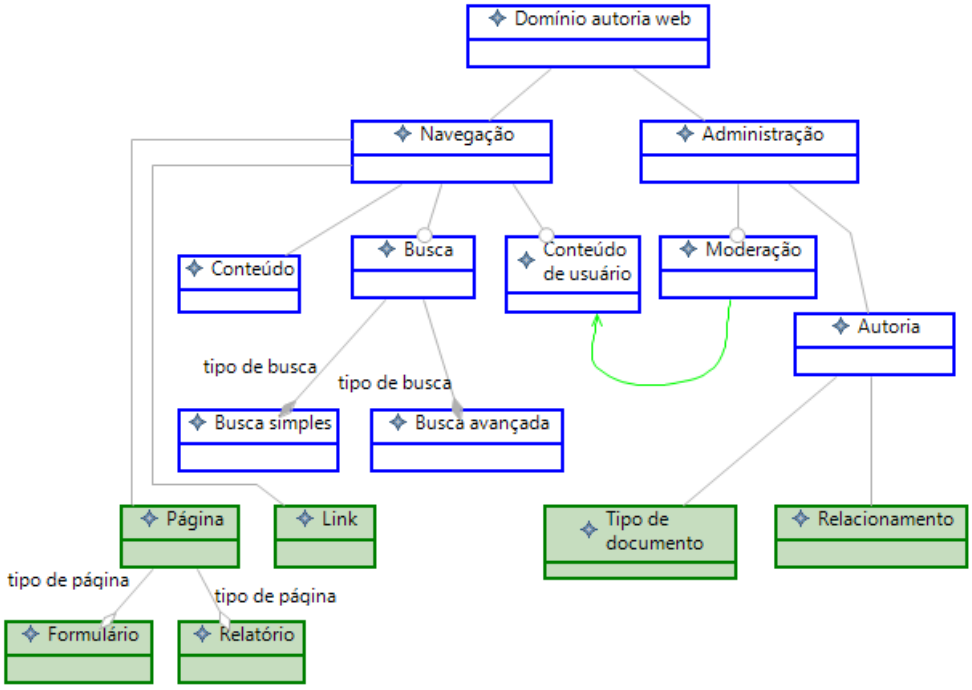


Figura 25: Modelo de *features* do domínio web de autoria de conteúdo

D_3. **Procurar por máquinas de estados:** muitos subdomínios podem ser representados por máquinas de estados, como as telas de um jogo ou o comportamento de uma entidade. Se for este o caso, este subdomínio irá provavelmente requerer uma DSL (máquina de estados) para sua variabilidade.

D_4. **Procurar por estruturas hierárquicas e de contenção:** relacionamentos do tipo *parte-de* são comuns em um domínio. Apesar de normalmente poderem ser representados no

modelo de *features*, alguns relacionamentos *parte-de* exigem informações extras. Por exemplo, no subdomínio GUI, um painel pode conter um ou mais botões. Mas onde estes botões estão localizados? Quais são seus tamanhos, cores e eventos associados? É possível incluir atributos em modelos de *features* (CZARNECKI; HELSEN; EISENECKER, 2004b) para expressar estas informações. Mas em termos do poder expressivo e facilidade de compreensão por parte dos especialistas, uma DSL seria mais adequada neste caso.

D_5**. Tentar o caminho mais fácil primeiro:** quanto mais próximo um subdomínio estiver do lado da configuração de rotina do espectro de variabilidade, mais fácil será a implementação de uma linguagem e transformações para o mesmo. Sempre que houver dúvida com relação à caracterização da variabilidade no subdomínio, o caminho mais fácil deve ser preferido, isto é, com um *wizard* ou configuração de *features*. Se estes se mostrarem insuficientes, então uma DSL mais complexa pode ser desenvolvida.

A saída desta atividade é a caracterização do tipo de variabilidade inerente a cada subdomínio. Mais importante, começa-se a identificar quais subdomínios irão requerer uma DSL dedicada.

Atividade ID.2. Definição das DSLs e do suporte ferramental (top-down)

Papéis: implementador do domínio, Especialista do domínio

Entradas: *PT.3.Validado. Modelagem do domínio*, *PT.4.Validado. Candidatos a subdomínio* e *PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios*, *PT.9.Avaliado. Projeto do domínio*, *PT.10. Subdomínios caracterizados*

Saídas: *PT.11.Inicial. Linguagens específicas de domínio*, *PT.12.Inicial. Suporte ferramental para DSLs*

Descrição: dependendo do tipo de variabilidade para cada subdomínio, a DSL associada será mais ou menos complexa. Em casos de variabilidade mais simples, baseada em *features*, a DSL pode ser composta por símbolos que representam *features* individuais, para indicar sua presença ou ausência. Czarnecki, Helsen e Eisenecker (2004a) propõem um método para derivar uma gramática livre de contexto para um modelo de *features*. Este método pode ser utilizado para criar uma DSL capaz de descrever todos os tipos de variabilidade característica a um modelo de *features*. Um gerador de *parsers* ou um *framework* de DSLs pode ser utilizado para criar o suporte à linguagem mais facilmente, suprindo a necessidade de ferramentas.

Em casos de variabilidade mais complexa, a DSL deve definir quais conceitos podem ser utilizados, como eles se relacionam entre si, e possíveis restrições que possam existir. Isto pode

ser realizado exclusivamente de forma *top-down*. Porém, a DSL deve também ser capaz de produzir modelos que sirvam de entrada para transformadores e geradores, o que inclui muitos detalhes que são específicos à plataforma e arquitetura escolhidas. É muito difícil perceber tais detalhes sem investigação mais aprofundada na implementação, e portanto uma abordagem *bottom-up* é utilizada para refinar esta DSL inicial. Esta atividade corresponde à parte *top-down* do desenvolvimento da DSL.

Conforme discutido na Seção 3.1.2, uma DSL pode ser textual (programas) ou visual (diagramas), e é normalmente composta de três elementos: a sintaxe abstrata, a sintaxe concreta e a semântica. Esta atividade cuida do desenvolvimento das sintaxes abstrata e concreta da DSL, além de ferramentas que permitam a criação de instâncias (programas ou diagramas) da DSL.

Em DSLs textuais, as sintaxes abstrata e concreta são normalmente representadas através de regras léxicas e gramaticais. Em DSLs visuais, a sintaxe abstrata corresponde ao metamodelo (GUIZZARDI; PIRES; SINDEREN, 2002) e a sintaxe concreta corresponde aos aspectos visuais dos elementos, normalmente utilizando figuras, ícones, linhas, setas, entre outras representações gráficas.

Para representar corretamente sua variabilidade, um subdomínio pode exigir um sistema de conceitos (sintaxes abstrata e concreta) totalmente diferente da modelagem de *features*, possivelmente exigindo também uma ferramenta de modelagem mais complexa. Mas mesmo não sendo suficiente para identificar conceitos de uma DSL (TOLVANEN; KELLY, 2005), um modelo de *features* pode servir de ponto de partida (CZARNECKI, 2005), sendo posteriormente complementado com informações de outros artefatos, como a arquitetura do domínio e o conhecimento do especialista (TOLVANEN; KELLY, 2005).

O desenvolvimento de DSLs é considerado uma ciência à parte (CZARNECKI; EISENECKER, 2000), dada sua complexidade. Além disso, não é um processo muito previsível, pois exige criatividade (VISSER, 2007). Esta abordagem busca prover alguma orientação, através de cinco sub-atividades, apresentadas a seguir.

Sub-atividade ID.2.1. Identificação das *features* iniciais da DSL

O primeiro passo é identificar as *features* que irão dar início à formação da sintaxe abstrata da DSL. Como descrito na atividade ID.1, os serviços e funções do domínio são normalmente descritos em termos de *features* de tecnologia do domínio, e portanto estas são boas candidatas a fazer parte de uma DSL. Considere por exemplo o domínio web de autoria de conteúdo mostrado na Figura 26A. Neste domínio, dois subdomínios podem ser identificados: navegação (Conteúdo, Busca, Busca simples, Busca avançada, Conteúdo de usuário, Página, Formulário,

Relatório e Link) e autoria (Autoria, Tipo de documento e Relacionamento). As *features* de tecnologia do domínio do subdomínio de navegação irão fazer parte da DSL de navegação (Páginas, Links, Formulários e Relatórios). Similarmente, a DSL do subdomínio de autoria irá incluir tipos de documentos e relacionamentos.

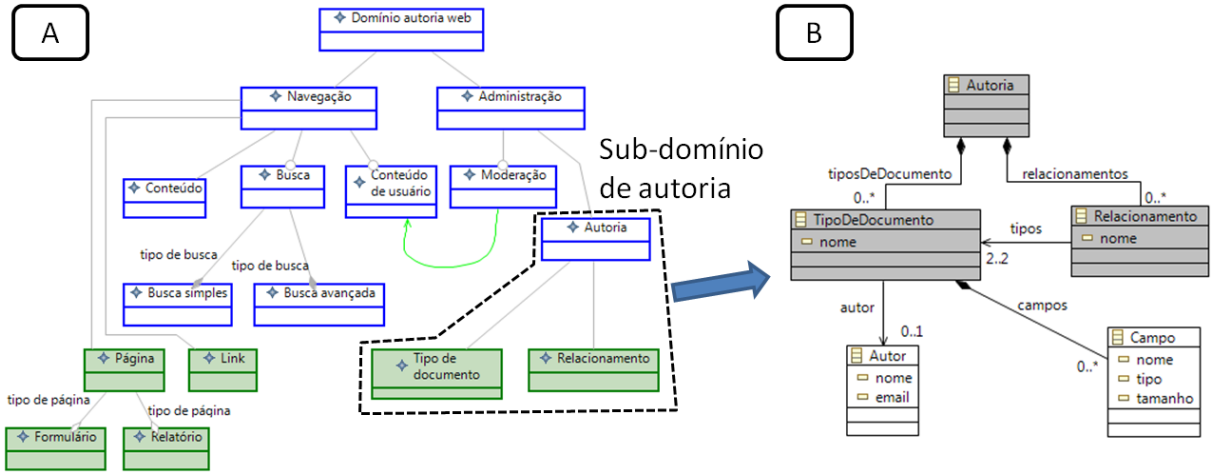


Figura 26: Definição do metamodelo da DSL de autoria Web

Sub-atividade ID.2.2. Definição da sintaxe abstrata

No segundo passo, as *features* identificadas são analisadas de forma mais aprofundada, para determinar como elas se relacionam entre si, e se conceitos adicionais são necessários. Estes conceitos adicionais são descritos em um metamodelo, que corresponde à sintaxe abstrata da DSL. Por exemplo, a Figura 26B mostra o metamodelo obtido para o subdomínio de autoria web. Elementos sombreados são derivados diretamente do modelo de *features*. Além das *features* *Autoria*, *Tipo de documento* e *Relacionamento*, este metamodelo contém os relacionamentos entre elas, e novos conceitos, como *Autor* e *Campo*.

Para auxiliar na definição do metamodelo, as seguintes regras podem ser utilizadas como guia:

- Uma *feature* (normalmente, uma *feature* de tecnologia de domínio) é mapeada em um conceito da DSL. Um conceito pode ser uma metaclasse em um metamodelo, caso se trate de uma DSL visual, ou uma regra de produção em uma gramática, caso se trate de uma DSL textual;

- *Sub-features* podem ser mapeadas como propriedades do conceito que as contém. Podem ser metaatributos em um metamodelo ou uma regra de produção ou atributo gramatical;

- *Sub-features* podem também ser mapeadas como conceitos separados, com relações *parte-de* adicionais sendo utilizadas para representar a contenção. Um relacionamento em um metamodelo corresponde a uma associação ou agregação, enquanto em linguagens textuais pode ser representado como uma regra de produção;
- Propriedades de conceitos podem precisar de tipos especiais do domínio, em vez dos tipos tradicionais como inteiro ou string. Por exemplo, *sub-features* opcionais ou do tipo *or* podem ser mapeadas como enumerações do domínio. Por outro lado, uma propriedade pode pertencer a tipos personalizados, como *Ponto* e *Tamanho*; e
- *Features* relacionadas podem ser conectadas por um conceito novo que descreve a relação, as cardinalidades, os conceitos participantes e seus papéis na relação. A análise de dependência de *features* (LEE; KANG, 2004) pode ser usada para identificar tais relações inicialmente, mas novas podem aparecer posteriormente.

Sub-atividade ID.2.3. Definição da sintaxe concreta

O terceiro passo corresponde à definição da sintaxe concreta, o que não é uma tarefa muito complexa. Em DSLs textuais externas (FOWLER, 2005), a sintaxe concreta é fortemente acoplada à sintaxe abstrata, aparecendo na forma de palavras reservadas e *tokens* descritos na gramática da linguagem. Se uma DSL interna (FOWLER, 2005) é utilizada, então as construções da linguagem que contém a DSL devem ser analisadas para determinar se podem representar os conceitos do domínio de forma adequada.

Em DSLs visuais, blocos decorados e linhas são a notação padrão. Formas geométricas específicas, como retângulos e elipses, ou imagens, podem também ser utilizadas para representar os conceitos da DSL de forma mais intuitiva e natural.

As seguintes regras podem ser utilizadas na criação desta notação:

- *Features* que são mapeadas para os conceitos do domínios podem ser representadas como blocos;
- Relacionamentos entre *features* podem ser representados como linhas;
- *Sub-features* que são mapeadas a valores booleanos podem ser representadas como decorações nos blocos (como uma borda diferenciada), atributos (como um ícone na frente de um elemento) ou linhas (como um losango em uma das terminações);
- *Sub-features* que são mapeadas a valores do tipo string podem ser representadas como decoradores textuais, como um texto dentro de um bloco ou sobre uma linha; e

- *Sub-features* que representam listas de itens podem ser representados como compartimentos em blocos (como os métodos e atributos da UML, por exemplo).

Sub-atividade ID.2.4. Definição da integração inter-DSL

O quarto passo lida com a integração entre diferentes DSLs, o que é um problema que surge quando se lida com múltiplos subdomínios. A integração inter-DSL deve ser gerenciada de forma que o desenvolvedor possa especificar modelos em ambas as linguagens utilizando conceitos que se relacionam. Por exemplo, no domínio web de autoria de conteúdo, uma página de formulário (do subdomínio de navegação) pode se referir a um tipo de documento (do subdomínio de autoria).

Referências baseadas em nome ou pontes entre modelos (WARMER; KLEPPE, 2006) são mecanismos simples que resolvem estes problemas. Referências baseadas em nome consistem em um simples atributo do tipo string, na DSL que contém a referência, que aponta para o nome de um elemento na DSL sendo referenciada. As checagens de tipo e integridade referencial devem ser feitas manualmente. Pontes entre modelos não são muito mais poderosas, consistindo na criação de um elemento, na DSL que contém a referência, que é uma cópia exata de um elemento na DSL sendo referenciada. Esta técnica propicia a checagem de tipos, mas a checagem de integridade referencial ainda precisa ser realizada manualmente.

Caso necessário, uma solução mais poderosa é apresentada por Hessellund, Czarnecki e Wasowski (2007), que propõem o uso de regras lógicas para estabelecer relações inter-DSL. Desta forma, consultas de mais alto nível podem ser realizadas, facilitando a checagem da consistência.

Sub-atividade ID.2.5. Construção da ferramenta de modelagem específica de domínio

Uma vez que as sintaxes abstratas e concretas estejam definidas, uma ferramenta de modelagem específica para a DSL é construída. *Frameworks* de DSLs, como GMF ou *openArchitectureWare*, entre outros, são a tecnologia mais apropriada para a implementação da ferramenta, uma vez que eles exigem pouco conhecimento na construção de linguagens para se alcançar resultados práticos rapidamente. Porém, DSLs mais complexas podem exigir uma participação mais ativa de um especialista em linguagens.

Esta última sub-atividade também inclui a definição de validações para capturar erros semânticos durante a modelagem, orientando o desenvolvedor na criação de diagramas ou programas segundo a DSL em questão (VÖLTER, 2008). Por exemplo, uma validação semântica

pode garantir que o comportamento de uma entidade em um jogo, modelada através de uma DSL visual, sempre tenha um comportamento padrão definido.

Após este quinto passo, obtém-se um conjunto de DSLs e ferramentas que permitem que um desenvolvedor represente diferentes tipos de variabilidade em cada subdomínio identificado, desde casos mais simples, baseada em *features*, até a variabilidade mais complexa. Mas as DSLs ainda não estão completas, pois até agora somente uma abordagem *top-down* foi utilizada. Podem existir detalhes adicionais que precisam ser incluídos antes que a DSL sirva de entrada para transformações e geração de código. É aqui que uma abordagem *bottom-up* entra em cena.

Atividade ID.3. Desenvolvimento das transformações e refinamento das DSLs (bottom-up)

Papéis: implementador do domínio, Especialista do domínio

Entradas: *PT.3.Validado. Modelagem do domínio*, *PT.4.Validado. Candidatos a subdomínio*, *PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios*, *PT.9.Avaliado. Projeto do domínio*, *PT.10. Subdomínios caracterizados*, *PT.11.Inicial. Linguagens específicas de domínio*, *PT.12.Inicial. Suporte ferramental para DSLs*

Saídas: *PT.11.Refinado. Linguagens específicas de domínio*, *PT.12.Refinado. Suporte ferramental para DSLs*, *PT.13. Transformações do domínio*, *PT.14. Implementação de referência*

Descrição: considerando-se somente seu poder representativo, uma DSL é composta somente pela sintaxe (abstrata e concreta). A sintaxe abstrata é voltada para interpretadores automáticos, enquanto a sintaxe concreta é projetada para ser usada por seres humanos, na criação de instâncias da DSL (diagramas ou programas).

Mas para ser útil no cenário do MDD, um terceiro elemento deve estar presente: a *semântica*, que define o significado dos elementos da linguagem. No contexto do MDD, a semântica é definida em forma de ações a serem executadas por um interpretador automático, que traduz o modelo (programa ou diagrama) em outra linguagem bem conhecida (KLEPPE, 2007).

No contexto da engenharia de domínio, a semântica adquire ainda outra importância, associada à variabilidade do domínio. Transformações de software são responsáveis por traduzir a variabilidade expressa em uma DSL em código concreto, de forma que o software gerado implemente as variantes selecionadas corretamente.

Até o momento, uma abordagem *top-down* foi seguida, utilizando modelos de alto nível, como o modelo de *features*, para produzir um conjunto inicial de DSLs. Estas DSLs são capazes de representar diferentes tipos de variabilidade em diferentes subdomínios identificados durante a análise.

Agora, uma abordagem *bottom-up* é adotada, utilizando o projeto através de exemplos (VARRÓ, 2006; WIMMER et al., 2007; ROBBES; LANZA, 2008) para produzir transformações e possivelmente refinar as DSLs iniciais. Partir de uma instância concreta de como o código deve ser é mais fácil do que identificar todos os detalhes a partir de uma perspectiva de mais alto nível (ROBBES; LANZA, 2008).

Esta atividade é realizada em quatro passos, apresentados a seguir.

Sub-atividade ID.3.1. Desenvolvimento da implementação de referência

O primeiro passo consiste no desenvolvimento de uma implementação de referência (MUSZYNSKI, 2005). Esta implementação de referência irá assumir dois papéis: primeiro, irá servir como um *framework* do domínio, encapsulando parte das comunalidades e oferecendo suporte a parte das variabilidades no nível de implementação, através de mecanismos como aqueles apresentados na Seção 3.2.1 e padrões como aqueles discutidos na atividade PD.3, da fase de projeto. A existência de um *framework* facilita a geração de código, uma vez que o gerador não precisa gerar todo o código, mas somente o código necessário para "completar" o *framework*. Para essa tarefa, o implementador do domínio, com a ajuda do especialista do domínio, segue a arquitetura definida na fase de projeto, utilizando as táticas e padrões associados para produzir uma implementação que irá dar suporte ao desenvolvimento subsequente.

Porém, nem toda comunalidade/variabilidade estará contida no *framework*. Assim, o segundo papel da implementação de referência é prover exemplos concretos das variabilidades restantes, servindo como um "retrato" do código que as transformações e geradores de código devem produzir, completando assim o suporte automatizado à variabilidade no domínio.

Para isso, a implementação de referência deve incluir exemplos de todos os pontos comuns e variáveis do domínio. Isto é mais fácil de ser alcançado para a variabilidade baseada em *features*, que é finita. As seguintes diretrizes podem ser utilizadas com este objetivo:

D_1. Começar implementando um exemplo completo que contém todas as *features* mandatórias, e uma seleção arbitrária de uma *feature* em cada grupo de *or features*. Deixar todas as *features* opcionais não-implementadas.

D_2. Para cada *feature* opcional, modificar o exemplo, criando uma nova versão do mesmo, de forma que a *feature* esteja presente. Caso algum dos padrões de projeto descritos no capítulo anterior tenha sido utilizado neste ponto do projeto para implementar esta variante em particular, basta realizar uma simples seleção da variante conforme previsto pelo padrão. Porém, pode ser que neste momento da implementação seja identificado um detalhe não previsto durante o projeto. Assim, deve-se também avaliar a aplicação de um novo padrão neste ponto para facilitar sua seleção no futuro.

D_3. Para *features* alternativas, modificar o exemplo para considerar a presença de cada alternativa separadamente, e em diferentes combinações. O uso de padrões destacado pela diretriz D_2 também deve ser considerado para este tipo de *feature*.

D_4. Prestar atenção às dependências entre as *features*. Por exemplo, se uma *feature* A depende de outra *feature* B, não faz sentido implementar A sem a presença de B.

D_5. Adotar uma abordagem de configuração em estágios (CZARNECKI; HELSEN; EISENECKER, 2004b), tentando eliminar as variabilidades em uma sequência lógica, considerando primeiro as *features* mais comuns, para reduzir o número de possibilidades e o número de versões produzidas a cada refinamento.

D_6. Dar maior atenção às variabilidades que possuem maior impacto na arquitetura, aquelas que são mais frequentes, e aquelas que exigem mais esforço para implementar manualmente.

D_7. Para reduzir o número de versões da implementação de referência e facilitar o seu gerenciamento, pode-se implementar múltiplas variantes simultaneamente, desde que não interfiram uma com a outra.

D_8. Nem todas as possibilidades precisam ser exploradas e implementadas no início. Algumas podem ser postergadas para quando a implementação do domínio estiver mais avançada, ou mesmo após o domínio estar em produção há algum tempo.

D_9. Nem toda variabilidade precisa ser incluída na implementação de referência, para fazer parte de uma DSL. Algumas variabilidades podem ser deixadas de fora, sendo necessária a sua implementação manual, por ocorrerem muito raramente ou exigirem muito esforço para automatizá-las.

D_{10}. Se existirem uma ou mais aplicações do domínio disponíveis, estas podem (e devem) ser consultadas durante o desenvolvimento da implementação de referência. Pode-se inclusive reutilizar trechos de código ou o suporte a alguma variabilidade que venha a estar já presente em alguma aplicação.

Para variabilidades baseadas em DSL, mais complexas, esta tarefa é mais difícil, já que

o número de combinações de elementos do domínio é literalmente infinito. É necessário mais conhecimento do especialista para se produzir exemplos representativos. As seguintes diretrizes podem ajudar:

D_{11}. Tentar explorar o espaço de variabilidade de duas maneiras: utilizando aplicações reais para derivar exemplos concretos da variabilidade, e também com extremos, como escolhas e combinações incomuns dos elementos da DSL.

D_{12}. Procurar popular atributos multivalorados e listas com mais de uma instância, caso contrário pode-se não perceber a existência deste tipo de atributo mais tarde, durante a atividade de mapeamento. Por exemplo, se uma DSL prevê a especificação de várias entidades de dados, deve-se criar duas ou três entidades de exemplo, e não somente uma.

D_{13}. Explorar as diferentes dimensões e tipos de atributos. Deve-se usar tanto valores pequenos e grandes como exemplos de atributos.

D_{14}. Resgatar produtos e aplicações analisados durante a análise de domínio, de modo a explorar a maneira como estes lidam com a variabilidade aberta. Deve-se utilizar este conhecimento como instâncias a serem cobertas pela implementação de referência.

D_{15}. Caso algum tipo de variabilidade seja muito complexa para ser automatizada através do MDD, deve-se então ao menos garantir que o código gerado dê suporte a algum tipo de mecanismo que permita a sua implementação manual. Deve-se experimentar estes mecanismos na implementação de referência.

Sub-atividade ID.3.2. Inspeção de código e mapeamento para elementos das DSLs

O segundo passo consiste na inspeção do código da implementação de referência em busca de trechos que correspondam a elementos de alguma DSL do domínio. O objetivo é identificar principalmente a presença de variantes no código, e mapeá-las para as DSLs correspondentes.

A inclusão de uma variante no código normalmente resulta em modificações em diferentes artefatos. Há quatro tipos de modificações que podem co-existir em um domínio (ANASTASOPOULOS; GACEK, 2001):

- **Adição de novas funcionalidades**: refere-se à inclusão de novos componentes, classes, funções, estruturas de dados ou trechos de código. Esta é a chamada variabilidade positiva. Por exemplo, no domínio web, a inclusão da *feature* de busca pode resultar em um novo componente, uma caixa de texto na página principal e novas estruturas no banco de dados;

- **Remoção de funcionalidades**: se refere à remoção de componentes, classes, funções, estruturas de dados ou trechos de código. É a chamada variabilidade negativa. Por exemplo, no domínio web, a inclusão da *feature* de busca pode resultar na remoção de um *banner* de propaganda do site, por falta de espaço;

- **Substituição de funcionalidades**: se refere à substituição de componentes, classes, funções, estruturas de dados ou trechos de código. Pode-se considerar como uma combinação de variabilidades negativa e positiva. Por exemplo, a inclusão da variante de busca avançada requer a substituição do componente de busca simples por um outro que implementa um algoritmo avançado; e

- **Plataforma/ambiente:** quando a inclusão da variante requer modificações na plataforma ou ambiente de execução. Por exemplo, no domínio web, a inclusão da *feature* de busca pode requerer a inclusão de uma biblioteca específica, uma versão diferente do banco de dados, ou mesmo uma máquina virtual diferente.

Esta atividade cuida da identificação exata destas alterações. Essencialmente, esta é uma atividade semelhante ao processo de comparação entre duas versões de um software que acontece em sistemas de controle de versões como CVS ou SVN. Compara-se cada exemplo que introduz uma variação com o exemplo original, registrando-se todas as alterações, obtendo-se um delta que representa a inclusão da variante.

Na prática, porém, a busca por tais alterações exige um conhecimento aprofundado do código e do domínio. A melhor maneira de encontrá-las é manter o registro das mudanças (ROBBES; LANZA, 2008) à medida que são feitas nos exemplos construídos durante o desenvolvimento da implementação de referência, por meio de um documento separado, ou mesmo com comentários e anotações no próprio código.

Cada trecho modificado do código precisa ser mapeado em pelo menos uma variante descrita em uma DSL ou modelo de *features*. Se existir uma modificação, mas não existirem elementos em uma DSL que a descrevam, então a DSL provavelmente precisa ser refinada (sub-atividade ID.3.3).

Em alguns casos, uma modificação pode aparentemente ser mapeada para diferentes elementos de uma ou mais DSLs, o que significa que esta modificação pode ser causada por diferentes variantes simultaneamente. Se for este o caso, deve-se tomar cuidado especial na implementação do suporte a estes pontos de variação quando ambos forem selecionados ao mesmo tempo.

Finalmente, pode acontecer de muitas modificações, em várias partes da implementação de

referência, serem mapeadas a um mesmo ponto de variação, o que significa que este elemento da DSL está espalhado em diferentes partes do código. Caso existam muitas modificações deste tipo, pode ser que a atividade de projeto não tenha produzido uma arquitetura adequada, e o uso de padrões de projeto pode reduzir este espalhamento. Ou então, este pode se tratar de uma preocupação ortogonal, caso em que técnicas da orientação a aspectos podem ajudar (VÖLTER; GROHER, 2007).

Sub-atividade ID.3.3. Refinamento das DSLs

Durante a inspeção de código, o implementador do domínio pode descobrir que mais detalhes ou informações precisam ser incluídos em uma DSL original antes que a mesma possa servir de entrada para transformações e geradores de código. Por exemplo, uma instância concreta de uma página web pode revelar detalhes como a disposição visual dos elementos, diferentes escolhas de elementos de entrada, que podem ter passado despercebidos durante o desenvolvimento da DSL inicial. Além disso, a inspeção pode identificar novos pontos de variação que não foram previstos durante a análise inicial. Finalmente, a inspeção pode encontrar erros e inconsistências com os conceitos iniciais e as variabilidades definidas na abordagem *top-down*.

Nestes casos, uma ou mais DSLs precisam ser refinadas, para introduzir novos elementos, modificar ou remover elementos existentes. Podem ser necessárias alterações na sintaxe abstrata, quando o refinamento for mais conceitual, alterações na sintaxe concreta, quando o refinamento envolver a representação física dos conceitos, ou alterações no suporte ferramental produzido. Deve-se também tomar cuidado para que não sejam introduzidas inconsistências, principalmente quando a DSL sendo refinada possui interações com outras DSLs e outros artefatos do domínio. Por isso, deve-se retornar à atividade ID.2 e refazer seus passos, visando manter o refinamento consistente.

Sub-atividade ID.3.4. Desenvolvimento das transformações e geradores de código

Com o refinamento das DSLs, o implementador constrói geradores de código baseados em *templates* (Seção 3.2.2). Para isso, ele realiza um processo conhecido como migração de código, onde o código da implementação de referência é anotado com marcações especiais, como *tags* e *scriptlets*, que fazem a associação entre o código e a DSL. Cada trecho de código correspondente a alguma variabilidade, identificado na sub-atividade ID.3.2, recebe algum tipo de anotação.

Tags normalmente dão suporte a comandos mais simples do tipo condicional ou iterativo.

Por exemplo, pode-se embutir o código referente a uma *feature* opcional dentro de uma *tag* do tipo condicional, que aceita como parâmetro um tipo booleano em uma DSL que indica a presença ou ausência da *feature*. Especificando-se este parâmetro com o valor *VERDADEIRO* faz com que o gerador inclua aquele trecho de código, enquanto o valor *FALSO* faz com que o gerador não o inclua. Exemplos de *tags* são aquelas incluídas pelo projeto JET (*Java Emitter Templates*), descrito na Seção 2.2.2.

Variabilidades mais complexas podem precisar de mecanismos mais flexíveis do que as *tags* para serem implementadas e parametrizadas. Neste caso, pode-se utilizar *scriptlets*, trechos de metacódigo que fazem um trabalho mais elaborado de cópia/colagem, produzindo uma saída que une fragmentos de código de forma a implementar a variabilidade exatamente como originalmente idealizada.

Geradores baseados em *templates* representam transformações modelo-para-texto, em um único passo. Porém, transformações modelo-para-modelo devem ser utilizadas para produzir geradores mais bem organizados. Nestes casos, cuidado especial deve ser tomado para que não seja necessário modificar os modelos intermediários, visando evitar problemas de inconsistência. Pode-se evitar este tipo de problema através de alguns padrões e práticas de sucesso na construção de geradores (VÖLTER, 2008; VÖLTER; BETTIN, 2004).

O processo de migração de código deve sempre manter a implementação de referência consistente com o código gerado, ou seja, deve ser possível gerar uma nova implementação de referência sempre que desejado, utilizando os geradores construídos. A princípio, não existe nenhum problema, desde que a migração de código não modifique a implementação de referência. Mas esta situação pode acontecer, já que durante a migração de código pode-se identificar oportunidades para melhorar o suporte à variabilidade.

Por exemplo, a implementação de referência pode implementar alternativas diferentes para um ponto de variação através de trechos de código distintos. Após a migração de código para o *template*, o implementador pode perceber que a criação de funções separadas, uma para cada alternativa, é uma solução melhor. Assim, ele modifica o *template* para implementar esta solução, fazendo com que a implementação de referência se torne inconsistente.

Mesmo que a migração de código não modifique a implementação de referência, a própria evolução do domínio normalmente causa este tipo de inconsistência. Sempre que for necessário corrigir algum erro, seja no gerador ou em outro lugar, deve-se idealmente realizar a manutenção primeiro na implementação de referência, testá-la e validá-la, e somente então repetir o processo de migração, de forma que a inconsistência não exista. Mas na prática, principalmente correções menores ou de erros do próprio gerador, modifica-se diretamente os *templates*, causando

inconsistência.

Para lidar com estes problemas, o seguinte processo é aplicado quando é necessário realizar alguma alteração (Figura 27).

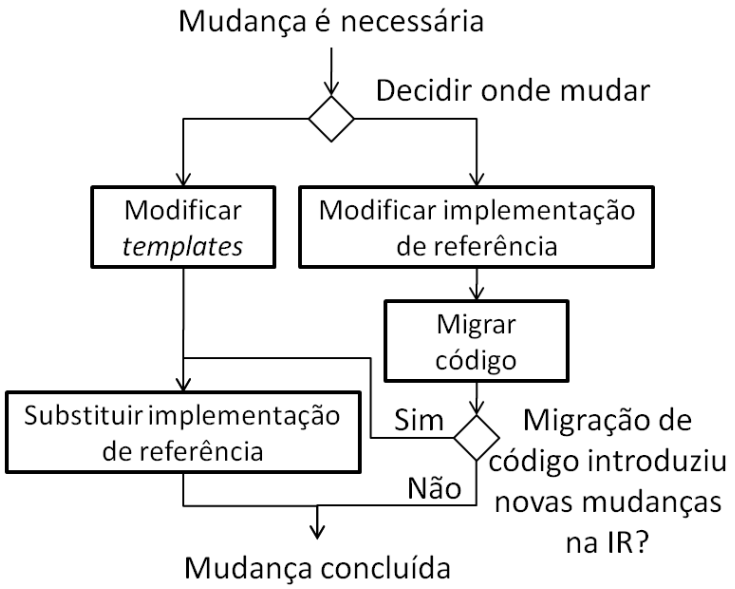


Figura 27: Manutenção da consistência da implementação de referência

Sempre que for necessária uma mudança, faz-se uma análise para decidir onde realizá-la:

- Se a mudança for realizada na implementação de referência, a migração de código é repetida, propagando as mudanças até os *templates*. Se a migração de código não introduzir ainda mais modificações, nada precisa ser realizado. Caso contrário, a implementação de referência precisa ser substituída, e a maneira recomendada é gerar uma nova implementação e utilizá-la como substituta; e
- Se a mudança for realizada direta nos *templates*, a implementação de referência precisa ser substituída, gerando-a novamente.

Atividade ID.4. Desenvolvimento do *framework* do domínio

Papéis: implementador do domínio, Especialista do domínio

Entradas: *PT.14. Implementação de referência*

Saídas: *PT.15. Framework do domínio*

Descrição: as atividades anteriores desenvolveram as DSLs, ferramentas e transformações do domínio. Para isso, utilizou-se uma implementação de referência como base da

identificação das variabilidades no código final. Porém, conforme discutido na atividade ID.3, a implementação de referência também serve como base para o desenvolvimento de um *framework* do domínio. Nesta atividade, portanto, o objetivo é transformar o restante da implementação de referência em um *framework* reutilizável.

De acordo com as características essenciais de um *framework*[1], a implementação de referência está relativamente próxima de se tornar um *framework* de domínio, pois:

- As classes da implementação de referência encapsulam conhecimento sobre um domínio de problema;
- Ela foi desenvolvida com base em uma arquitetura bem definida a partir de um processo centrado no suporte à variabilidade e com o objetivo de implementar diferentes diretrizes arquiteturais;
- A implementação de referência não define somente as classes de forma isolada, mas também as interfaces e conexões entre as mesmas, em uma estrutura que representa parte do conhecimento daquele domínio; e
- A implementação de referência possui suporte para pontos fixos e pontos variáveis, a exemplo de um *framework*, que podem ser estendidos e instanciados por um desenvolvedor de aplicações;

Portanto, para transformar a implementação de referência em um *framework* de domínio, a princípio só é necessário tornar explícito os pontos variáveis, em termos de código não-gerado, uma vez que o restante das variabilidades será implementada somente pelos transformadores e geradores de código.

O código não-gerado, idealmente, já foi estruturado utilizando padrões de software que facilitam a seleção e implementação de algumas variabilidades, na atividade PD.3. Desta forma, aqui é necessário especificar as interfaces resultantes da aplicação dos padrões. Por exemplo, caso tenha sido utilizado o padrão *Decorator* para *features* alternativas complementares, aqui são definidas as interfaces de cada alternativa, com seus métodos sendo descritos individualmente.

Pode ser necessário o desenvolvimento de algum mecanismo para facilitar a instanciação dos pontos variáveis. Por exemplo, pode-se criar um objeto *Singleton* para facilitar a instanciação das *features* implementadas utilizando-se o padrão *Abstract Factory*. Pode-se

[1] Uma discussão mais aprofundada sobre *frameworks* e seu papel na reutilização de software é apresentada no Apêndice A.

inclusive criar um método separado para cada *sub-feature* alternativa, de forma que para escolher uma destas, basta chamar o método correspondente, ao invés de instanciar a fábrica desejada.

Finalmente, a implementação de referência pode ser complementada com componentes não incluídos originalmente, por não terem sido serem necessários no momento ou por uma decisão estratégica.

A saída desta atividade é uma versão revisada e completada da implementação de referência, preparada para ser mais reutilizável e passível de instanciação no desenvolvimento de aplicações. Uma vez pronto, o *framework* do domínio toma o lugar da implementação de referência.

Atividade ID.5. Documentação do domínio

Papéis: implementador do domínio, Especialista do domínio

Entradas: *PT.11.Refinado. Linguagens específicas de domínio*, *PT.12.Refinado. Suporte ferramental para DSLs*, *PT.13. Transformações do domínio*, *PT.15. Framework do domínio*

Saídas: *PT.16. Documentação do domínio*

Descrição: documentação de software pode reduzir tempo e esforço no desenvolvimento de novos projetos, facilitar o porte de aplicações para outras plataformas, além de ajudar usuários a compreender um software mais facilmente (PHOHA, 1997). No contexto da reutilização de software, a documentação é ainda mais importante (SILVA; WERNER, 1996; KOTULA, 1998; TAULAVUORI; NIEMELA; KALLIO, 2004), uma vez que um reutilizador precisa identificar, compreender, adaptar e integrar componentes de software em suas aplicações. Além disso, normalmente componentes de software são reutilizados por equipes que não conhecem nem possuem contato com os desenvolvedores (SZYPERSKI; GRUNTZ; MURER, 2002).

Porém, não existe um padrão universalmente reconhecido para documentação de software, em parte porque estilos e conteúdo de documentação diferem entre programadores, e alguma vezes diferem para um mesmo programador em circunstâncias diferentes. Adicionalmente, a escolha da linguagem de programação e a natureza de um programa podem ditar um estilo particular de documentação que pode não ser facilmente aplicável a outro ambiente (VOTH, 2004).

Existem alguns padrões e formatos voltados à documentação de software em geral (PHOHA, 1997), como o padrão *ANSI/ANS 10-3-1995 para documentação de software* e o RAS (*Reusable*

Asset Specifications ou Especificações de Artefatos Reutilizáveis) (VOTH, 2004). Estes podem ser úteis para a documentação de diferentes tipos de software. Com base nestes e em outros trabalhos relacionados à documentação de componentes (SILVA; WERNER, 1996; KOTULA, 1998; TAULAVUORI; NIEMELA; KALLIO, 2004; BASILI; ABD-EL-HAFIZ, 1996), a presente abordagem propõe um formato específico para a documentação, além de alguns princípios:

P_1. **Hipertexto**. O hipertexto mostrou-se uma técnica eficiente para a documentação. Por exemplo, ajuda a aumentar o conhecimento que engenheiros de software adquirem ao percorrer repositórios de componentes de software (ISAKOWITZ; KAUFFMAN, 1996). Também facilita o processo de navegação através da informação, além de ser um formato já familiar à maioria dos usuários. Assim, sempre que possível, a documentação deve ser apresentada como hipertexto.

P_2. **Comentários embutidos no código-fonte**. A documentação deve estar sempre consistente e atualizada com relação ao código-fonte. Embutindo-se a documentação no código-fonte, é mais fácil atualizá-la sempre que forem feitas mudanças no código, já que ambos estão no mesmo lugar;

P_3. **Automação**. No contexto do MDD, é possível gerar, junto com código, diferentes tipos de artefatos, incluindo documentação. Assim, sempre que possível, deve-se automatizar a geração de documentação para livrar o desenvolvedor deste tipo de trabalho manual;

P_4. **Utilize a semântica da linguagem**. Muitas construções da própria linguagem de programação contém informação útil à reutilização de software. Por exemplo, a comparação de assinatura (ZAREMSKI; WING, 1995) analisa a própria definição das interfaces para determinar se um componente pode ser reutilizado em um determinado contexto. Também é possível, por exemplo, determinar algumas características de um componente examinando-se o código-fonte, até mesmo de forma automática. Assim, a documentação deve utilizar todo tipo de informação semântica disponível no próprio código-fonte.

P_5. **Diagramas e figuras**. A documentação deve incluir diagramas e figuras sempre que possível. No MDD, muitas das informações visuais estão disponíveis na própria linguagem específica do domínio. Assim, um artefato que serve de entrada para um gerador é o próprio documento visual do software a ser gerado.

Para a documentação de artefatos de código, pode-se utilizar um formato como o descrito por Almeida et al. (2008), que contempla cinco seções e seus campos relacionados: **informações básicas**, contendo descrições gerais sobre o componente, como seu nome, propósito e palavras-chave; **descrição detalhada**, contendo a definição das interfaces e informações sobre a linguagem de implementação, versionamento, etc; **informações de qualidade**, descrevendo as informações necessárias para a garantia de qualidade do

componente, tais como o pacote de testes; **informações sobre implantação**, tais como as dependências necessárias para compilar e implantar o componente; e **informações de suporte**, com um guia de instalação e contatos que ajudam na identificação das pessoas responsáveis pelo suporte técnico do componente.

Para outros artefatos específicos do MDD, os seguintes formatos podem ser utilizados:

- Para DSLs, deve-se incluir informações sobre:
    1. O subdomínio com o qual a DSL se relaciona;
    2. A gramática ou metamodelo utilizado;
    3. Detalhes sobre a sintaxe concreta, tais como o significado das figuras, linhas, ícones, decorações;
    4. Ferramentas de modelagem que podem ser utilizadas;
    5. Outras DSLs que interagem com esta;
    6. Transformações/geradores com os quais a DSL se relaciona (isto é, serve de entrada); e
    7. Exemplos de sua utilização.
- Para ferramentas de modelagem, deve-se incluir informações sobre:
    1. O subdomínio com o qual a ferramenta se relaciona;
    2. A DSL para a qual a ferramenta oferece suporte;
    3. Detalhes sobre o seu desenvolvimento, tais como as tecnologias ou *frameworks* utilizados;
    4. Manual de instalação e utilização; e
    5. Contatos para suporte técnico.
- Para transformações de software, deve-se incluir informações sobre:
    1. O raciocínio por trás das regras de transformação, uma vez que as linguagens de transformação nem sempre são intuitivas o suficiente para serem compreendidas sem informação adicional;
    2. Metamodelos ou DSLs de origem e destino;
    3. Detalhes sobre o seu desenvolvimento, tais como as linguagens, tecnologias ou *frameworks* utilizados; e

4. Especificação dos parâmetros necessários para a transformação.

- Para geradores de código, deve-se incluir informações sobre:

1. Comentários, no próprio código dos geradores, sobre os mapeamentos com as DSLs de entrada;
2. Metamodelos ou DSLs de origem;
3. Linguagens de destino;
4. Descrição e raciocínio do código gerado;
5. Detalhes sobre o seu desenvolvimento, tais como as linguagens, tecnologias ou *frameworks* utilizados;
6. Especificação dos parâmetros necessários para a geração de código.

É importante ressaltar que a maioria das informações necessárias para esta documentação foi sendo produzida ao longo do processo de engenharia de domínio. Por exemplo, os metamodelos das DSLs, o mapeamento entre DSLs e código, cruzamento entre diferentes DSLs, entre outras informações, foram produzidos durante o processo de análise, projeto e implementação. Neste caso, trata-se somente de um processo de empacotamento da informação, e possivelmente algum refinamento. Em outros casos, como a descrição e raciocínio do código gerado, especificação dos parâmetros para as transformações e geradores de código, as informações precisam se definidas e detalhadas somente após sua conclusão.

7.3 Considerações finais

A fase de implementação do domínio é quando as informações coletadas sobre o domínio e modeladas durante a análise e projeto são concretizadas em forma de artefatos de implementação. Assim, ao final desta atividade, tem-se um conjunto de artefatos reutilizáveis, ou componentes, que implementam parte da arquitetura do domínio, linguagens específicas para os subdomínios, ferramentas que permitem criar modelos para estes subdomínios e transformações modelo-modelo e modelo-texto capazes de gerar código específico para a arquitetura do domínio.

O Quadro 12 resume as atividades de implementação do domínio.

O Quadro 13 descreve os produtos de trabalho da fase de implementação do domínio.

Atividades e sub-atividades	**Entradas**	**Saídas**
ID.1. Caracterização da variabilidade dos subdomínios	PT.3.Validado. Modelagem do domínio PT.4.Validado. Candidatos a subdomínio PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios PT.9.Avaliado. Projeto do domínio	PT.10. Subdomínios caracterizados
ID.2. Definição das DSLs e do suporte ferramental (top-down) ID.2.1. Identificação das *features* iniciais da DSL ID.2.2. Definição da sintaxe abstrata ID.2.3. Definição da sintaxe concreta ID.2.4. Definição da integração inter-DSL ID.2.5. Construção da ferramenta de modelagem específica de domínio	PT.3.Validado. Modelagem do domínio PT.4.Validado. Candidatos a subdomínio PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios PT.9.Avaliado. Projeto do domínio PT.10. Subdomínios caracterizados	PT.11.Inicial. Linguagens específicas de domínio PT.12.Inicial. Suporte ferramental para DSLs
ID.3. Desenvolvimento das transformações e refinamento das DSLs (bottom-up) ID.3.1. Desenvolvimento da implementação de referência ID.3.2. Inspeção de código e mapeamento para elementos das DSLs ID.3.3. Refinamento das DSLs ID.3.4. Desenvolvimento das transformações e geradores de código	PT.3.Validado. Modelagem do domínio PT.4.Validado. Candidatos a subdomínio PT.5. Histórico de decisões sobre inclusão/exclusão de subdomínios PT.9.Avaliado. Projeto do domínio PT.10. Subdomínios caracterizados PT.11.Inicial. Linguagens específicas de domínio PT.12.Inicial. Suporte ferramental para DSLs	PT.11.Refinado. Linguagens específicas de domínio PT.12.Refinado. Suporte ferramental para DSLs PT.13. Transformações do domínio PT.14. Implementação de referência
ID.4. Desenvolvimento do *framework* do domínio	PT.14. Implementação de referência	PT.15. Framework do domínio
ID.5. Documentação do domínio	PT.11.Refinado. Linguagens específicas de domínio PT.12.Refinado. Suporte ferramental para DSLs PT.13. Transformações do domínio PT.15. Framework do domínio	PT.16. Documentação do domínio

Quadro 12: Resumo das atividades para implementação do domínio orientada a modelos

No próximo capítulo é apresentado um possível modelo de ciclo de vida para a utilização da abordagem, considerando-se um processo evolutivo e interativo, com características mais próximas à realidade das organizações.

Produto de trabalho	**Descrição**	**Estado**
PT.10. Subdomínios caracterizados	Definição do tipo de variabilidade característico de cada subdomínio: baseada em *features* ou baseada em DSLs	Nenhum
PT.11. Linguagens específicas de domínio	Definição das sintaxes abstrata e concreta das linguagens específicas de domínio para os subdomínios identificados durante o processo. A sintaxe abstrata das DSL visuais normalmente é um metamodelo, enquanto a sintaxe abstrata das DSLs textuais é uma gramática	1. *Inicial*: versão da DSL produzida somente através de uma abordagem *top-down*. Normalmente faltam detalhes que só serão identificados após a implementação 2. *Refinado*: versão inicial da DSL refinada após uma abordagem *bottom-up*, que identifica mais detalhes para a linguagem
PT.12. Suporte ferramental para DSLs	Ferramentas de modelagem para as DSLs. Podem ser ferramentas visuais, para a criação de diagramas segundo uma DSL visual, ou ferramentas textuais, para a criação de programas segundo uma DSL textual	1. *Inicial*: versão das ferramentas produzidas somente através de uma abordagem *top-down*. Normalmente faltam detalhes que só serão identificados após a implementação 2. *Refinado*: versão inicial das ferramentas após uma abordagem *bottom-up*, que identifica mais detalhes para a linguagem
PT.13. Transformações do domínio	Transformações modelo-para-modelo e modelo-para-código para serem utilizadas em conjunto com as DSLs do domínio	Nenhum
PT.14. Implementação de referência	Uma implementação do domínio contendo exemplos das diferentes variabilidades identificadas durante o processo	Nenhum
PT.15. Framework do domínio	Conjunto de classes reutilizáveis de um domínio, estruturadas de modo a formar um esqueleto de uma aplicação do domínio, com os pontos variáveis bem definidos e mecanismos que possibilitam a sua instanciação	Nenhum
PT.16. Documentação do domínio	Documentação dos diferentes artefatos de implementação do domínio, incluindo componentes, DSLs, ferramentas, transformações e geradores de código	Nenhum

Quadro 13: Descrição dos produtos de trabalho da fase de implementação do domínio

8 *Avaliação*

A combinação entre desenvolvimento orientado a modelos e reutilização de software tem como promessa a reutilização em alto nível, como defendido pelos pesquisadores (NEIGHBORS, 1980; KRUEGER, 1992; GRISS, 1995; FRAKES; ISODA, 1994; JACOBSON; GRISS; JONSSON, 1997) desde as primeiras discussões sobre reutilização de software. Diversos pesquisadores, entre eles Czarnecki et al. (2005), concordam que MDD e reutilização são esforços complementares. As próprias origens destas duas linhas de pesquisa estão interligadas, conforme discutido no Capítulo 9.

A presente tese buscou investigar de forma mais aprofundada esta idéia: defende-se que a combinação entre o desenvolvimento orientado a modelos e reutilização de software em um processo sistemático exige o gerenciamento dos múltiplos subdomínios que podem existir em um domínio, de modo a oferecer um grau de automação adequado. Para isso, a preocupação com o MDD deve estar presente em todas as etapas do processo, incluindo a análise, o projeto e a implementação do domínio.

A avaliação desta tese consistiu na realização de estudos empíricos envolvendo a aplicação da abordagem em projetos de engenharia de domínio, com o objetivo de fornecer evidências ou indícios sobre sua validade.

É importante ressaltar que a avaliação realizada tem caráter mais exploratório, com alguns pontos que ainda precisam ser melhorados para que os resultados possam ser mais confiáveis, conforme discutido na Seção 8.6. Mas de qualquer forma, puderam ser extraídas algumas conclusões, como apresentado no final deste capítulo.

A seguir são apresentados os detalhes sobre a avaliação realizada.

8.1 Definição

Para a definição dos estudos desta avaliação, foi utilizado o paradigma GQM (*Goal-Question Metric*) (BASILI; CALDIERA; ROMBACH, 1994). Neste paradigma, devem ser

definidos os objetivos da avaliação, que devem ser rastreados a um conjunto de dados que definem estes objetivos de forma operacional, e a interpretação destes dados com respeito aos objetivos definidos. O rastreamento entre os objetivos e os dados é feito através de questões que caracterizam os objetivos de forma mais específica.

Assim, seguindo o formato sugerido no GQM, são definidos dois objetivos para esta avaliação, assim descritos e detalhados por meio das respectivas questões específicas:

G_1. ***Analisar*** *a abordagem orientada a modelos para reutilização de software* ***com o objetivo de*** *determinar se ela promove aumento e/ou melhoria na reutilização de software, quando comparada com o desenvolvimento não orientado a modelos,* ***com respeito aos*** *artefatos do domínio produzidos* ***do ponto de vista do*** *pesquisador* ***no contexto de*** *projetos de engenharia de domínio.*

- Q_1. Analisando-se um mesmo projeto desenvolvido com e sem a abordagem, é possível observar um aumento e/ou melhoria na reutilização de software no projeto que utilizou a abordagem?
- Q_2. Os artefatos de software produzidos com a abordagem são mais reutilizáveis do que aqueles produzidos em uma abordagem não orientada a modelos?

G_2. ***Analisar*** *a abordagem orientada a modelos para reutilização de software* ***com o objetivo de*** *determinar a sua importância em todo o ciclo de vida* ***com respeito aos*** *benefícios obtidos e dificuldades de utilização* ***do ponto de vista do*** *pesquisador* ***no contexto de*** *projetos de engenharia de domínio.*

- Q_3. Os participantes que utilizaram a abordagem perceberam, durante as atividades da abordagem referentes à preocupação com MDD desde o início do desenvolvimento (fase de análise), algum benefício para a implementação dos artefatos do MDD (DSLs, transformações e geradores de código)?
- Q_4. Os participantes que utilizaram a abordagem tiveram dificuldades que causaram prejuízo ao desenvolvimento, em termos de atrasos e curva de aprendizado?

O objetivo G_1 diz respeito à tese de que o MDD oferece meios concretos para que a reutilização de conhecimento possa ocorrer em maior grau e de forma mais adequada, quando comparada com um processo não orientado a modelos. Assim, a questão Q_1 busca observar este aumento e/ou melhora na reutilização comparando-se dois projetos: um desenvolvido sem a abordagem e outro desenvolvido com a abordagem. Contudo, mesmo que não seja observado efetivamente um aumento conforme as métricas especificadas, isto não significa

que a abordagem não favoreça mais reutilização. Existe a possibilidade de os artefatos serem mais reutilizáveis quando desenvolvidos com a abordagem, mesmo que não tenham sido efetivamente reutilizados no seu maior potencial nestes estudos. Assim, a questão Q_2 busca avaliar este aspecto.

O objetivo G_2 diz respeito à tese de que a utilização do MDD exige uma preocupação que deve estar presente em todo o ciclo de vida, devendo ser tratada de forma consistente desde a análise até a implementação. Assim, a questão Q_3 se refere à obtenção de algum benefício observável com a aplicação da abordagem desde o início. A questão Q_4 busca identificar se a utilização do MDD desde o início traz mais problemas do que benefícios.

Nesta avaliação, o projeto desenvolvido sem a abordagem consistiu na aplicação dos conceitos de reutilização de software de forma ad hoc. Foram produzidos componentes de software reutilizáveis e uma arquitetura de referência para o domínio, porém sem a execução de atividades específicas de um método voltado à reutilização. Nos três casos, a implementação obtida pelo desenvolvimento sem a abordagem foi aproveitada durante a atividade de desenvolvimento da implementação de referência. Desta forma, o código final obtido com e sem a abordagem foi praticamente o mesmo.

8.1.1 Coleta de dados

Para a obtenção de dados que possam conter indícios ou evidências sobre estas questões, foram definidas formas qualitativas e quantitativas de coleta de dados, combinando a percepção subjetiva dos envolvidos nos estudos empíricos com medidas referentes à estrutura do software produzido e outros atributos de qualidade. Foram definidas formas de coleta de dados específicas para cada questão relacionada aos objetivos da avaliação, conforme apresentado a seguir.

Questão Q_1. Analisando-se um mesmo projeto desenvolvido com e sem a abordagem, é possível observar um aumento e/ou melhoria na reutilização de software no projeto que utilizou a abordagem?

É difícil definir o que significa “aumento e/ou melhoria na reutilização”. A reutilização ocorre de diversas maneiras, e dependendo do contexto, pode significar diferentes benefícios. Por exemplo, a reutilização obtida por meio de herança é completamente diferente da reutilização obtida com um gerador de código, sendo difícil determinar qual das duas oferece maiores benefícios, pois não há métricas consistentes capazes de medir ambos os aspectos de forma normalizada.

Existem diferentes métricas voltadas à reutilização[1], que focam em aspectos econômicos e estruturais de artefatos de software. No contexto desta questão destacam-se apenas os aspectos estruturais, pois esta tese trata de uma abordagem voltada para a engenharia para reutilização. A literatura apresenta algumas métricas simples para medir a reutilização no nível estrutural dos artefatos, mas nenhuma delas é capaz de oferecer uma imagem completa do nível de reutilização em um sistema (MASCENA; ALMEIDA; MEIRA, 2005). Assim, faz-se necessária uma análise mais cuidadosa, combinando-se diferentes métricas que avaliem os aspectos importantes para este estudo específico.

Assim, as seguintes métricas foram definidas para a questão Q_1:

M_1. *Porcentagem de reutilização (RP - Reuse Percent).* Esta é a métrica mais básica de reutilização, sendo definida como a razão entre o número de linhas de código reutilizadas e o número total de linhas de código (POULIN; CARUSO, 1993). Um cuidado especial deve ser tomado com esta métrica, pois pode levar a conclusões incorretas. Por exemplo, caso seja reutilizado um único método, pequeno, de uma classe com milhares de linhas de código, esta porcentagem seria alta mas não refletiria a realidade de que apenas uma porção da classe foi reutilizada. A coleta desta métrica envolve a identificação dos módulos reutilizados e a contagem das linhas de código dos módulos reutilizados e não-reutilizados.

M_2. *Razão de reutilização (RR - Reuse Ratio).* Esta métrica é calculada da mesma forma que a M_1, porém também considerando-se a reutilização do tipo caixa-branca (DEVANBU et al., 1996): artefatos modificados até um certo nível são considerados como reutilizados. Devanbu et al. (1996) sugerem o valor de 25% como margem de tolerância: artefatos que tiveram mais de 25% de seu código modificado não são considerados como reutilizados.

M_3. *Porcentagem de reutilização não desejada.* Conforme destacado na métrica M_1, a reutilização de grandes trechos de código que não são efetivamente utilizados pode distorcer a métrica de porcentagem de reutilização. O uso desta métrica permite determinar se este efeito está ocorrendo. Além disso, reutilizar código que não é efetivamente aproveitado pode ser prejudicial, agregando informações descartáveis que podem confundir a equipe durante a manutenção do software. Portanto, esta métrica também ajuda a caracterizar a reutilização. Esta métrica consiste no cálculo da porcentagem, para cada artefato reutilizado, das linhas de código que não são efetivamente utilizadas em relação ao número total de linhas de código reutilizadas. Para a coleta desta métrica pode-se utilizar funcionalidades das IDEs que permitem determinar quais métodos não são chamados, por exemplo.

M_4. *Porcentagem de reutilização gerada.* Esta métrica calcula a porcentagem de artefatos

[1] Uma revisão sobre métricas para MDD e reutilização é apresentada no Capítulo 9.

produzidos por geração automática e que foram reutilizados, em relação ao total de reutilização. É calculada como sendo a porcentagem entre o número de linhas de código geradas e o número de linhas de código reutilizadas. Permite caracterizar a reutilização que está ocorrendo em domínios implementados através do MDD.

M_5. *Razão entre especificação e código*. Esta métrica é complementar à anterior, e permite determinar a relação entre um elemento de especificação e o código gerado correspondente. Por exemplo, se a especificação de uma classe, com atributos e relacionamentos, produz 1000 linhas de código, tem-se um maior número de linhas de código reutilizadas do que a especificação de um arquivo de configuração com 10 linhas que produz somente 20 linhas de código. Esta métrica é calculada da seguinte forma:

$$REC = \sum LOC(módulosgerados) : \sum NEE(modelos)$$

onde NEE(modelo) corresponde ao número de elementos de especificação do modelo.

Um elemento de especificação pode ser uma linha de texto (em uma DSL textual) ou um elemento gráfico de um diagrama, seja ele uma caixa, atributo, linha ou outro elemento gráfico similar. Para efeito de comparação, considera-se que uma linha textual é aproximadamente equivalente a um elemento gráfico, pois apesar de o elemento gráfico ser mais simples de criar e editar do que uma linha de texto, ele normalmente possui propriedades que precisam ser configuradas individualmente.

As métricas M_4 e M_5 presumem a existência de geração de código, e portanto não podem ser utilizadas para comparar um projeto desenvolvido sem a abordagem (que não possui geração de código) com um projeto desenvolvido com a abordagem. Porém, elas são úteis para ajudar a caracterizar a reutilização que está ocorrendo com a abordagem.

Dois pontos de discussão sobre estas métricas precisam ser destacados:

1. No contexto envolvendo geração de código, pode-se argumentar que o uso do tamanho em linhas de código nas métricas pode distorcer os resultados, uma vez que geradores de código normalmente produzem código mais denso (MODELWARE, 2006c). No entanto, vale ressaltar que nesta abordagem a construção dos geradores é feita a partir de uma implementação de referência, ou seja, o código gerado segue os mesmos padrões, convenções e formato utilizados pelos programadores humanos. Neste contexto, a comparação é válida (MODELWARE, 2006c); e

2. Estas métricas são aplicadas somente ao código-fonte, não sendo úteis para elementos

de modelagem. Como o objetivo é comparar a quantidade de conhecimento que foi reutilizado na construção do software final, a comparação dos códigos é razoável, uma vez que o tamanho de um módulo reflete de forma indireta a quantidade de conhecimento ali encapsulado.

Esse segundo ponto só é válido para a questão Q_1. Na questão Q_2, discutida a seguir, os artefatos do MDD, como modelos, transformações e geradores de código precisam ser avaliados quanto aos seus atributos de qualidade relacionados à reutilização, e portanto devem ser incluídos nas medições.

Questão Q_2. Os artefatos de software produzidos com a abordagem são mais reutilizáveis do que aqueles produzidos em uma abordagem não orientada a modelos?

Conforme discutido anteriormente, as métricas para reutilização M_1, M_2 e M_3 referentes à questão Q_1 apresentam problemas por não considerar a natureza dos artefatos reutilizáveis e nem a maneira com que estes são reutilizados, penalizando, por exemplo, grandes sistemas e sistemas pouco modularizados (monolíticos) (MASCENA; ALMEIDA; MEIRA, 2005; MASCENA, 2007; ALMEIDA et al., 2007a).

Por este motivo, existe outra vertente que defende a idéia de que é melhor tentar medir a reutilização através da avaliação de atributos de qualidade que influenciam a reusabilidade de um determinado artefato de software. Neste sentido, são sugeridas métricas indiretas, como manutenibilidade e complexidade (POULIN, 1994). Estas medidas talvez ofereçam uma visão melhor sobre os reais benefícios da abordagem, já tendo sido utilizadas com sucesso em outros estudos relacionados à reutilização de software, como por exemplo o trabalho de Almeida et al. (2007a). Assim, as seguintes métricas foram definidas para esta questão:

M_6. *Instabilidade de módulo.* De acordo com Martin (1994), o que torna um software rígido, frágil e difícil de ser reutilizado é a interdependência entre seus módulos. Um software é rígido se ele não puder ser facilmente modificado, isto é, uma única mudança inicia uma cascata de mudanças em diferentes módulos. Além disso, quando a extensão da mudança não pode ser prevista pelo projetista, seu impacto não pode ser estimado, o que dificulta a estimativa de custos, tornando o software rígido. A métrica de instabilidade de um módulo (I) busca avaliar esta característica do software, sendo definida da seguinte maneira:

$$I = \frac{AE}{AA + AE}$$

Onde AA significa *Acoplamento Aferente*, ou seja, o número de classes fora deste módulo que dependem de classes neste módulo, e AE significa *Acoplamento Eferente*, ou seja, o número

de classes dentro deste módulo que dependem de classes fora deste módulo.

A métrica de instabilidade varia entre 0 e 1, onde I próximo a 0 indica um módulo estável e I próximo a 1 indica um módulo instável. Como não existem muitos dados prévios para serem utilizados como base para esta medida, neste estudo, considerou-se que módulos com instabilidade < 0,5 são considerados mais estáveis do que a média, a exemplo do estudo de Almeida et al. (2007a).

Esta métrica pode ser utilizada tanto em artefatos de código-fonte como em artefatos do MDD, como DSLs, modelos específicos de domínio, transformações e geradores, já que se baseia no conceito de dependências. Porém, enquanto existem ferramentas automatizadas para extrair tais valores diretamente do código-fonte, a coleta com relação aos demais artefatos deve ser manual.

M_7 Complexidade ciclomática. Um aspecto crítico na Engenharia de Software se relaciona à separação de interesses e à modularização de um sistema de software com o objetivo de se obter módulos bem definidos, testáveis e de fácil manutenção. Durante a década de 70, McCabe (1976) desenvolveu uma técnica matemática que provê uma base quantitativa para modularização e identificação de módulos de software difíceis de testar ou manter. Nesta técnica, uma medida de complexidade é apresentada, tendo como objetivo medir e controlar o número de caminhos em um programa. No contexto da reutilização, a complexidade tem papel fundamental, pois artefatos mais complexos possuem menor facilidade de serem reutilizados (MASCENA, 2007; POULIN, 1994). A complexidade ciclomática é calculada a partir de um grafo conectado que representa o fluxo de execução de um módulo, da seguinte maneira:

$$CC(G) = E - N + p$$

onde E = número de arcos de um grafo, N = número de nós de um grafo e p = número de componentes conectados.

Para esta métrica, valores entre 1 e 10 indicam que se trata de um módulo simples, sem muito risco. Valores entre 11 e 20 representam módulos moderadamente complexos. Valores entre 21 e 50 representam alta complexidade, e valores maiores que 50 representam módulos completamente não-testáveis e com alto risco.

Por ser aplicável a grafos, esta métrica pode ser utilizada diretamente em modelos visuais do tipo grafo. Modelos textuais também podem ser transformados em grafos, analisando-se o metamodelo correspondente.

M_8. Índice de Manutenibilidade. A manutenibilidade é um atributo desejável tanto como

uma medida instantânea como uma previsão da manutenibilidade com o tempo. Esforços para medir e rastrear a manutenibilidade têm por objetivo reduzir ou reverter a tendência de degradação de integridade de um sistema, e indicar quando pode ser mais barato ou menos arriscado reconstruir do que modificar (VANDOREN, 1997). No contexto da engenharia de domínio, ela é um indicador importante da reusabilidade de um artefato (MASCENA; ALMEIDA; MEIRA, 2005), uma vez que um domínio está constantemente em evolução, com novas *features* ou produtos sendo desenvolvidos (ALMEIDA et al., 2007a). O Índice de Manutenibilidade (IM) busca medir a manutenibilidade de um módulo, sendo definido da seguinte maneira (VANDOREN, 1997):

$$IM = 171 - 5.2 * \ln(aveV) - 0.23 * aveV(g') - 16.2 * \ln(aveLOC) + 50 * \sin(\sqrt{2.4 * perCM})$$

onde: aveV = média do volume Halstead V por módulo, aveV(g') = média da complexidade ciclomática estendida por módulo, aveLOC = média de linhas de código por módulo, e perCM = média da porcentagem de linhas de comentário por módulo.

A complexidade ciclomática é discutida na métrica M_7. O volume Halstead V de um módulo é calculado da seguinte maneira:

$$V = N * \log_2 n$$

onde N = número total de operadores + número total de operandos do módulo e n = número total de operadores distintos + número total de operandos distintos do módulo.

Segundo esta métrica, módulos com IM menor do que 65 são difíceis de serem mantidos, módulos entre 65 e 85 possuem manutenibilidade razoável e módulos com IM maior do que 85 possuem boa manutenibilidade.

Esta métrica é normalmente utilizada em artefatos de código-fonte, mas também pode ser aplicada a geradores de código, uma vez que os mesmos também possuem operadores e operandos. Geradores baseados em *templates*, que são um dos focos desta abordagem, possuem código embutido e marcações especiais que representam operações simples, como condições e laços. As seguintes regras são utilizadas para cálculo do volume Halstead V de um gerador baseado em *template*:

- Variáveis, trechos de código contínuo e constantes são consideradas **operandos**;
- Marcações responsáveis por uma consulta, como uma seleção de elementos de um modelo, ou impressão de um determinado valor, são consideradas **operadores**;

- Marcações condicionais e de laços são considerados **operadores**; e
- Trechos de código embutido são analisados da mesma forma que código-fonte tradicional.

Esta métrica não pode ser aplicada a modelos, uma vez que estes não necessariamente contêm elementos que podem ser associados com operandos e operadores. Nestes casos, devem ser utilizadas métricas específicas para modelos.

Conforme demonstrado por Genero et al. (2007) e por Genero, Poels e Piattini (2008), a manutenibilidade de um modelo é determinada principalmente por sua compreensibilidade. Neste sentido, os autores identificaram, no contexto da modelagem Entidade-Relacionamento e diagramas de classes, que as propriedades estruturais de um modelo que são relevantes para que um ser humano possa compreendê-lo facilmente são os atributos e relacionamentos 1:N. O tamanho de um modelo em termos do número de entidades não parece ser relevante para compreensibilidade. Assim, duas métricas foram definidas para avaliar a manutenibilidade dos modelos nesta abordagem. Apesar de originalmente terem sido desenvolvidas para modelos E-R e diagramas de classes, o conceito de atributos e relacionamentos está presente em várias linguagens do tipo grafo, e portanto podem ser aplicadas em outros tipos de modelos similares.

M_9. *Número de Atributos.* Nesta métrica, são contados os atributos em um modelo. Em modelos visuais (diagramas), um atributo é uma propriedade textual de um elemento visual (um ícone, uma caixa ou um nó de um grafo). Em modelos textuais, um atributo é uma propriedade textual de um conceito de primeira classe, conforme definido na gramática da linguagem.

M_{10}. *Número de Relacionamentos.* Nesta métrica, são contados os relacionamentos em um modelo. Em modelos visuais (diagramas), um relacionamento é representado por uma linha entre dois elementos, ou outra representação relacional entre dois ou mais elementos, como por exemplo a representação de conjuntos no GME (*Generic Modeling Environment* - Seção 2.2.2). Em modelos textuais, um relacionamento consiste em uma referência entre dois conceitos de primeira classe, conforme definido na gramática da linguagem.

Essas duas métricas são absolutas, ou seja, sabe-se que quanto mais atributos e relacionamentos, menos compreensível será um modelo, mas não se tem uma medida para ser utilizada como parâmetro para determinar se um modelo é ou não compreensível, e por consequência possui alta manutenibilidade. Como valor de referência, foi utilizada nesta tese a média obtida em experimentos envolvendo diagramas E-R e UML, conforme descrito por Genero et al. (2007) e por Genero, Poels e Piattini (2008):

$$NA = \frac{42,63 + 40,22}{2} = 41,425$$

$$NR = \frac{13,13 + 8,88 + 8 + 8,2}{4} = 9,595$$

Portanto, truncando-se estes valores, considerou-se neste estudo que modelos com menos de 41 atributos e menos do que 9 relacionamentos possuem maior manutenibilidade do que a média.

As métricas M_9 e M_{10} permitem a avaliação tanto de modelos visuais e textuais, como também seus metamodelos, permitindo também avaliar a manutenibilidade das DSLs.

Questão Q_3. Os participantes que utilizaram a abordagem perceberam, durante as atividades da abordagem referentes à preocupação com MDD desde o início do desenvolvimento, algum benefício para a implementação dos artefatos do MDD (DSLs, transformações e geradores de código)?

Para avaliar se o uso da abordagem desde o início do desenvolvimento produz algum benefício, foi realizada uma entrevista com os participantes que utilizaram a abordagem, com perguntas que avaliaram se os benefícios almejados foram efetivamente percebidos e observados. Decidiu-se por uma entrevista ao invés de um questionário, pois os estudos não envolveram um número muito grande de participantes, e desta forma houve mais flexibilidade na observação dos resultados da aplicação da abordagem. A entrevista consistiu das perguntas a seguir, com respostas em aberto:

- O modelo de *features* ajudou na definição das linguagens específicas de domínio, transformações e geradores de código?
- A descrição da variabilidade em cenários (casos de mudança) facilitou a definição das linguagens específicas de domínio, transformações e geradores de código?
- A identificação de candidatos a subdomínio facilitou a identificação das áreas do domínio a serem automatizadas?
- A identificação de candidatos a subdomínio facilitou a definição das linguagens específicas de domínio, transformações e geradores de código?
- O processo investigativo baseado em decisões para inclusão/exclusão de subdomínios foi utilizado? Se sim, ele facilitou o processo de desenvolvimento dos artefatos do MDD?
- O uso das diretrizes e padrões arquiteturais específicos para reutilização e MDD facilitou o desenvolvimento dos artefatos do MDD e da arquitetura do domínio?

- A etapa de avaliação arquitetural ajudou a identificar falhas antes de a implementação ser iniciada?
- As diretrizes de implementação de DSLs, transformações e geradores de código facilitaram a implementação dos artefatos do MDD?
- O formato de documentação proposto foi adequado, auxiliando na descrição dos artefatos reutilizáveis desenvolvidos?
- Quais foram as vantagens de se utilizar a abordagem de MDD no desenvolvimento?
- Quais foram as desvantagens de se utilizar a abordagem de MDD no desenvolvimento?

Questão Q_4. Os participantes que utilizaram a abordagem tiveram dificuldades que causaram prejuízo ao desenvolvimento, em termos de atrasos e curva de aprendizado?

Para avaliar as dificuldades de utilização da abordagem, foram incluídas as seguintes perguntas, na entrevista realizada na questão Q_3, referente às dificuldades percebidas pelos participantes:

- Quais foram as dificuldades para o aprendizado da abordagem?
- Quais foram as dificuldades para definição do modelo de *features*?
- Quais foram as dificuldades para descrição da variabilidade em cenários (casos de mudança)?
- Quais foram as dificuldades para a identificação de candidatos a subdomínio?
- Quais foram as dificuldades para realizar o processo investigativo baseado em decisões para inclusão/exclusão de subdomínios?
- Cite outras dificuldades percebidas durante a utilização da abordagem de MDD desde o início do desenvolvimento.

8.1.2 Hipóteses

Como ensina Albert Einstein, físico alemão que viveu entre 1879 e 1955: *‘‘Nenhuma quantidade de experimentação pode provar que estou certo; um único experimento pode provar que estou errado”*. Com base nesta premissa científica, é comum trabalhar-se com a idéia de uma hipótese nula, ou seja, define-se como hipótese algo que o experimentador deseja rejeitar,

e projeta-se um ou mais experimentos que têm como objetivo testar esta hipótese (WOHLIN et al., 2000).

Assim, a hipótese nula para esta avaliação pode ser descrita com base nas questões e métricas definidas anteriormente:

H_{0a}: analisando-se um mesmo projeto desenvolvido com e sem a abordagem, as métricas de *porcentagem de reutilização* (M_1), *razão de reutilização* (M_2), *porcentagem de reutilização não-desejada* (M_3), *porcentagem de reutilização gerada* (M_4) e *razão entre especificação e código* (M_5), analisadas conjuntamente, não evidenciam nenhum aumento ou melhoria na reutilização de software no projeto que utilizou a abordagem.

H_{0b}: os artefatos de software produzidos com a abordagem não são mais reutilizáveis do que aqueles produzidos em uma abordagem não orientada a modelos

- (M_6) Instabilidade de módulo: $I_{comAbordagem} \geq I_{semAbordagem}$
- (M_7) Complexidade Ciclomática: $CC_{comAbordagem} \geq CC_{semAbordagem}$
- (M_8) Índice de Manutenibilidade: $IM_{comAbordagem} \leq IM_{semAbordagem}$
- (M_9) Número de Atributos: $NA \geq 41$
- (M_{10}) Número de Relacionamentos: $NR \geq 9$

H_{0c}: os participantes que utilizaram a abordagem não perceberam, com as atividades da abordagem referentes à preocupação com MDD desde o início do desenvolvimento, nenhum benefício para a implementação dos artefatos do MDD (DSLs, transformações e geradores de código).

H_{0d}: os participantes que utilizaram a abordagem tiveram dificuldades que causaram prejuízo ao desenvolvimento, em termos de atrasos e curva de aprendizado.

Para a parte H_{0a} desta hipótese não foram definidos itens individuais para cada métrica, pois as mesmas não podem ser avaliadas de forma isolada para determinar se houve de fato aumento na reutilização. Portanto, foi feita uma análise descritiva considerando-se todas as métricas em conjunto, buscando rejeitar a hipótese nula. Com relação à parte H_{0b}, foram definidas comparações contrárias ao que se deseja demonstrar, com exceção das métricas M_9 e M_{10}, pois no desenvolvimento tradicional (sem abordagem), não foram produzidos modelos, e portanto apenas a métrica M_8 foi considerada suficiente para avaliar a manutenibilidade. Assim, as comparações de M_9 e M_{10} foram feitas com os valores estabelecidos como referência. Para as partes H_{0c} e H_{0d} não foi definida nenhuma métrica, pois a análise foi qualitativa.

A rejeição da hipótese nula deve ser feita em favor de uma hipótese alternativa, que normalmente representa a negação da hipótese nula. Neste cenário, as seguintes alternativas são definidas:

H_{1a}: analisando-se um mesmo projeto desenvolvido com e sem a abordagem, as métricas de *porcentagem de reutilização* (M_1), *razão de reutilização* (M_2), *porcentagem de reutilização não-desejada* (M_3), *porcentagem de reutilização gerada* (M_4) e *razão entre especificação e código* (M_5), analisadas conjuntamente, possuem evidência ou indícios sobre o aumento e/ou melhoria no nível de reutilização de software no projeto que utilizou a abordagem.

H_{1b}: os artefatos de software produzidos com a abordagem são mais reutilizáveis do que aqueles produzidos em uma abordagem não orientada a modelos

(M_6) Instabilidade de módulo: $I_{comAbordagem} < I_{semAbordagem}$

(M_7) Complexidade Ciclomática: $CC_{comAbordagem} < CC_{semAbordagem}$

(M_8) Índice de Manutenibilidade: $IM_{comAbordagem} > IM_{semAbordagem}$

(M_9) Número de Atributos: $NA < 41$

(M_{10}) Número de Relacionamentos: $NR < 9$

H_{1c}: os participantes que utilizaram a abordagem perceberam, com as atividades da abordagem referentes à preocupação com MDD desde o início do desenvolvimento, algum benefício para a implementação dos artefatos do MDD (DSLs, transformações e geradores de código).

H_{1d}: os participantes que utilizaram a abordagem não tiveram dificuldades que causaram prejuízo ao desenvolvimento, em termos de atrasos e curva de aprendizado.

8.2 Descrição dos projetos utilizados nos estudos empíricos e sua execução

Foram realizados três estudos envolvendo a aplicação da abordagem em projetos de engenharia de domínio. Para o primeiro estudo, foi escolhido o domínio de aplicações de autoria de conteúdo para a Web, por se tratar de um domínio relativamente simples de ser implementado manualmente e pela disponibilidade de especialistas para eventuais consultas. Este primeiro estudo foi completamente desenvolvido a partir do zero para a avaliação, mas foi

o resultado da evolução dos estudos utilizados como base para testar e refinar as atividades da abordagem, conforme descrito na Seção 4.3.

O segundo estudo foi realizado na indústria, durante um estágio de doutorado, e a escolha do domínio envolvido, de aplicações distribuídas baseadas em computação em nuvem, não foi feita pelo autor desta tese, mas sim pelos líderes do grupo no qual o estágio foi realizado.

O terceiro estudo, também realizado na indústria, envolveu o domínio de eventos científicos, e a escolha se deu por conveniência e proximidade da empresa ao grupo de pesquisa.

Os estudos são comparativos, ou seja, visam comparar os resultados do desenvolvimento realizado sem a abordagem com o desenvolvimento realizado com a abordagem. Para essa comparação, foram utilizadas implementações de exemplo desenvolvidas antes do uso da abordagem, que não contemplavam modelos ou qualquer tipo de artefato relacionado ao MDD, e as implementações desenvolvidas com a abordagem, que incluem uma arquitetura preparada para o MDD, além dos artefatos específicos para este contexto, como DSLs, transformações e geradores de código.

A seguir são apresentados mais detalhes sobre cada estudo.

8.2.1 Autoria de conteúdo para a Web

O primeiro estudo foi realizado em ambiente acadêmico, e envolveu o domínio de aplicações de autoria de conteúdo para a Web. São aplicações onde um administrador publica conteúdo a ser visualizado por muitos usuários, como por exemplo portais de notícias, páginas pessoais, fóruns e *blogs*. Trata-se de um domínio técnico, envolvendo os requisitos de persistência e navegação na Web.

Neste estudo, as linguagens específicas de domínio, geradores, a arquitetura e a implementação de referência foram desenvolvidos a partir do zero, utilizando a abordagem definida nesta tese. Foi executado pelo autor desta tese, com a ajuda de especialistas de domínio. Inicialmente, foram realizados estudos sobre as tecnologias de implementação necessárias, e em seguida a abordagem foi aplicada no desenvolvimento dos artefatos do domínio.

Foi desenvolvida uma infraestrutura composta de artefatos de software gerados e não-gerados, em uma arquitetura em três camadas. Foram desenvolvidas quatro linguagens específicas de domínio: uma linguagem visual para persistência, uma linguagem textual para navegação, uma linguagem textual para a produção de relatórios e uma linguagem visual para configuração das *features* presentes no produto. Também foram desenvolvidas transformações e geradores de código que produzem aplicações completas, segundo as especificações dos

modelos.

O Quadro 14 mostra um resumo dos dados relevantes sobre este estudo.

Domínio	**Autoria de conteúdo para Web**
Local	**ICMC/USP - São Carlos/SP**
Participantes	**1 (autor)**
Tempo total	**3 meses (dedicação integral)**
Número de DSLs	**4 (persistência, navegação, relatórios e *features*)**
Número de artefatos de geração (transformações e geradores)	**38**
Tamanho total (LOC) artefatos de geração	**1610**
Tamanho total (LOC) implementação de referência	**5941**
Número de *features* do domínio	**15**
Tecnologias de implementação	**Apache Tomcat, MySQL, Java, JSP, JSTL, Servlets, XML, SQL, Eclipse (versão *Ganymede*) + WebTools plug-in**
Tecnologias MDD	**Eclipse (versão *Ganymede*) EMF (*Eclipse Modeling Framework*) GMF (*Graphical Modeling Framework*) JET (*Java Emitter Templates*) openArchitectureWare**

Quadro 14: Dados sobre o estudo envolvendo o domínio de autoria de conteúdo para Web

8.2.2 Aplicações distribuídas baseadas em computação em nuvem

O segundo estudo foi realizado em ambiente industrial, envolvendo o domínio de aplicações distribuídas baseadas em computação em nuvem (LUCRÉDIO; JACKSON; SCHULTE, 2008). As principais características deste domínio são o alto grau de incerteza sobre a topologia da aplicação, a heterogeneidade dos ambientes e infraestrutura e o alto grau de cooperação entre os componentes distribuídos. Este estudo foi realizado durante um estágio de doutorado realizado pelo autor da tese nos laboratórios da Microsoft Research, em Redmond, Estados Unidos (LUCRÉDIO; JACKSON; SCHULTE, 2008).

Este domínio já vinha sendo explorado pelos pesquisadores da Microsoft Research, que desenvolveram uma teoria baseada em modelagem de negócios e um conjunto de três linguagens específicas para este objetivo (JACKSON; SCHULTE, 2008): uma linguagem visual para descrever os dados do sistema, uma linguagem visual para descrever as características dos tipos de elementos distribuídos, sua conectividade e quais dados possuem, e uma linguagem visual para descrever a semântica das operações de manipulação dos dados. O aspecto mais interessante desta especificação é que os modelos de dados e operações não precisam se preocupar com a localização dos dados e nem que tipo de elemento distribuído é responsável por sua manutenção.

A teoria diz que é possível gerar código responsável pela execução distribuída das operações e verificação de integridade global do sistema, de acordo com regras específicas definidas nos modelos.

Trata-se de um domínio predominantemente técnico, envolvendo os aspectos de execução distribuída e computação em nuvem, mas com um componente de negócios, uma vez que as regras de negócio podem ser definidas utilizando a linguagem de modelagem de operações.

Neste estudo, realizado pelo autor da tese junto com um pesquisador da Microsoft Research, foi utilizada a abordagem definida nesta tese, porém com menor foco na análise, uma vez que já existiam versões iniciais das linguagens específicas de domínio. Após o estudo das tecnologias de implementação envolvidas, estas versões iniciais das linguagens foram refinadas, e passou-se para as fases de projeto e implementação do domínio, onde foram desenvolvidos, a partir do zero:

- A arquitetura para o domínio;
- Uma implementação de referência baseada em uma infraestrutura responsável por algumas das funções básicas de comunicação e distribuição; e
- Um conjunto de transformações e geradores que produzem uma aplicação completa no topo da infraestrutura.

O Quadro 15 resume alguns dados relevantes sobre este estudo.

8.2.3 Eventos científicos

O terceiro estudo foi realizado também em ambiente industrial, e envolveu o domínio de eventos científicos. Este domínio engloba sistemas de submissão e inscrição em eventos, mala direta, gerenciamento de secretaria e gerenciamento de feiras. Trata-se de um domínio de negócios, e a reutilização envolve principalmente regras de negócio relacionadas ao gerenciamento de eventos científicos.

Este estudo foi feito em uma empresa situada em São Carlos que trabalha com uma linha de produtos deste domínio, realizando desenvolvimento e customização dos produtos do domínio, muitas vezes vendendo os mesmos de forma integrada. O gerenciamento da variabilidade nesta linha é realizado com a ajuda de alguns componentes reutilizáveis, mas principalmente utilizando técnicas de gerenciamento de configuração. Assim, cada sistema vendido produz diferentes versões que implementam as diferentes variabilidades do domínio. A reutilização

Domínio	**Aplicações distribuídas baseadas em computação em nuvem**
Local	**Microsoft Research - Redmond/WA - USA**
Participantes	**2 (incluindo o autor)**
Tempo total	**3 meses (dedicação integral)**
Número de DSLs	**3 (dados, conectividade e operações)**
Número de artefatos de geração (transformações e geradores)	**29**
Tamanho total (LOC) artefatos de geração	**2847**
Tamanho total (LOC) implementação de referência	**12127**
Número de *features* do domínio	**-**
Tecnologias de implementação	**Microsoft Visual Studio 2008, Microsoft SQL Server, C#, .NET, .NET Remoting, Microsoft Volta, PNRP (*Peer Name Resolution Protocol*) DBML (DataBase Modeling Language), Microsoft LINQ to SQL**
Tecnologias MDD	**Microsoft Visual Studio 2008 Microsoft Text Templates GME (*Generic Modeling Environment*)**

Quadro 15: Dados sobre o estudo envolvendo o domínio de aplicações distribuídas baseadas em computação em nuvem

ocorre recuperando-se uma determinada versão que seja próxima dos requisitos do cliente, e modificando-a para satisfazer aos requisitos.

Após o contato inicial, foi estabelecido que o estudo seria realizado como um projeto paralelo na empresa, por uma equipe composta de dois funcionários com dedicação parcial, com o auxílio e orientação do autor desta tese. Como neste estudo o autor não participaria do desenvolvimento, foi necessário um período de treinamento, realizado da seguinte forma:

- Em um primeiro momento, um dos participantes do projeto realizou um curso de 20 horas sobre desenvolvimento orientado a modelos, oferecido no ICMC/USP São Carlos e ministrado pelo autor desta tese[2];
- Em seguida, o autor da tese realizou um treinamento interno de 4 horas com os participantes, apresentando as atividades e detalhes da abordagem; e
- O autor da tese disponibilizou como exemplo os documentos e artefatos produzidos durante o estudo do domínio Web, que ficou à disposição da equipe para consulta.

Após este período, o autor desta tese permaneceu disponível para sanar dúvidas e responder questionamentos durante todo o projeto, uma vez que se tratam de conceitos novos para a empresa e os participantes.

[2]O material deste curso encontra-se disponível no endereço `http://www.icmc.usp.br/~lucredio/downloads/files/cursoMDD/`.

Após o treinamento, a equipe utilizou a abordagem para realizar a análise, projeto e implementação do domínio. Uma vez que já existia a linha de produtos implantada na empresa, o projeto arquitetural se resumiu principalmente à organização da infraestrutura de geração de código de forma integrada aos artefatos existentes na linha de produtos. E na implementação do domínio, foi utilizado um produto concreto como implementação de referência.

Como resultado, foram definidas duas linguagens específicas de domínio: uma linguagem textual para a definição das *features*, e a sua configuração através de atributos específicos para a linha de produtos, e uma linguagem textual para a definição de formatos de certificados, um subdomínio identificado pela equipe durante a análise. Foram também construídos os geradores responsáveis por configurar novos produtos e produzir código customizado.

O Quadro 16 resume alguns dados relevantes sobre este estudo.

Domínio	**Eventos científicos**
Local	**Aptor Consultoria e Desenvolvimento de Software Ltda.**
Participantes	**2**
Tempo total	**2 meses (dedicação parcial)**
Número de DSLs	**2 (configuração e certificados)**
Número de artefatos de geração (transformações e geradores)	**8**
Tamanho total (LOC) artefatos de geração	**1375**
Tamanho total (LOC) implementação de referência	**71873**
Número de *features* do domínio	**29**
Tecnologias de implementação	**Adobe DreamWeaver, PHP, MySQL**
Tecnologias MDD	**Eclipse (versão *Ganymede*)** **EMF (*Eclipse Modeling Framework*)** **JET (*Java Emitter Templates*)**

Quadro 16: Dados sobre o estudo envolvendo o domínio de eventos científicos

8.3 Coleta dos dados

A coleta das métricas foi realizada com o auxílio de diferentes ferramentas. Para os projetos baseados em código Java, foram utilizadas as ferramentas *Eclipse Metrics*[3] (Instabilidade e Complexidade Ciclomática) e *JHawk*[4] (Índice de Manutenibilidade). Estas ferramentas trabalham com código-fonte em Java. Para arquivos JSP, foi utilizado o código Java de seus *Servlets* correspondentes. Os *templates* do *JET* são convertidos em Java, e portanto estas ferramentas também puderam ser utilizadas.

[3] `http://metrics.sourceforge.net/`

[4] `http://www.virtualmachinery.com/jhawkprod.htm`

Para código C#, não foi encontrada nenhuma ferramenta disponível capaz de calcular estas métricas. Porém, foi utilizada a ferramenta *Net2Java*[5], um *plug-in* do NetBeans capaz de converter código escrito em C# para Java. Como são linguagens bastante similares, a conversão é realizada de forma quase direta. Além disso, não foi necessário executar o código convertido. As métricas são estruturais, e se baseiam em dependência entre classes, operadores, operandos, estruturas de laços e condições, e outros elementos do código que são comuns a ambas linguagens. Assim, o único trabalho foi corrigir alguns erros da conversão, visando deixar o código Java compilável, requisito das ferramentas *Eclipse Metrics* e *JHawk*, e estas foram utilizadas para extrair as métricas.

O código PHP utilizado no terceiro estudo não é orientado a objetos, e não foi encontrada nenhuma ferramenta capaz de extrair estas métricas para este tipo de código, e nem ferramentas automatizadas de conversão. Uma vez que a quantidade de código deste projeto é muito grande, a coleta manual das métricas de Instabilidade, Complexidade Ciclomática e Índice de Manutenibilidade, que exigem uma inspeção detalhada do código, não foi realizada no terceiro estudo.

Para os demais artefatos, como metamodelos, modelos e geradores baseados em *templates* interpretados, como os *templates* do *openArchitectureWare*, foi necessário realizar o cálculo manual, como por exemplo a contagem de acoplamento aferente e eferente, análise de grafos dos programas e a contagem do número de atributos e relacionamentos em modelos. Como o número e tamanho destes artefatos é consideravelmente reduzido, este trabalho foi realizado sem dificuldades.

As análises qualitativas, através de entrevista, somente foram realizadas no terceiro estudo, uma vez que nos dois primeiros estudos o autor da tese participou do desenvolvimento.

Após a coleta dos dados, os mesmos foram analisados utilizando estatística descritiva através de gráficos do tipo *box plot* (FENTON; PFLEEGER, 1998), que permitem visualizar a dispersão e balanceamento das amostras. *Box plots* podem ser utilizados de diferentes formas (WOHLIN et al., 2000). Nesta tese, foi utilizada a abordagem definida por Fenton e Pfleeger (1998), que propõem a utilização do seguinte cálculo para análise das extremidades:

$$L_{inferior} = q_1 - (q_3 - q_1) * 1.5$$

$$L_{superior} = q_3 + (q_3 - q_1) * 1.5$$

onde q_1 e q_3 são o primeiro (25%) e terceiro (75%) quartis, respectivamente.

[5] `https://net2java.dev.java.net/`

Dado um conjunto de dados seguindo uma distribuição normal, teoricamente não deveriam ser encontrados elementos abaixo do limite inferior e acima do limite superior. Caso existam, estes devem ser analisados e deve-se decidir se serão incluídos ou excluídos da análise. Por exemplo, quando o elemento fora dos limites se tratar de código-fonte, deve-se analisar se ele segue os mesmos padrões e formatos utilizados na maioria do código. Se este elemento possuir muitas linhas de comentário ou se for um código reutilizado de outro projeto sem uma reformatação, ele pode não representar a realidade do código produzido, e deve ser excluído da análise. Por outro lado, podem existir elementos de código extremamente complexos ou extremamente simples, devido à natureza do próprio domínio. Nestes casos, o elemento deve ser mantido.

Para cada estudo, foram analisados somente os artefatos efetivamente utilizados pelo desenvolvedor. Ou seja, artefatos de código completamente gerados e que não precisam ser modificados ou manipulados pelo desenvolvedor não foram incluídos nas métricas. Porém, em alguns gráficos, o código gerado foi também analisado para propiciar melhor caracterização da reutilização que está ocorrendo.

A seguir são apresentados os resultados para cada estudo, de forma individual, junto com uma discussão geral sobre os resultados.

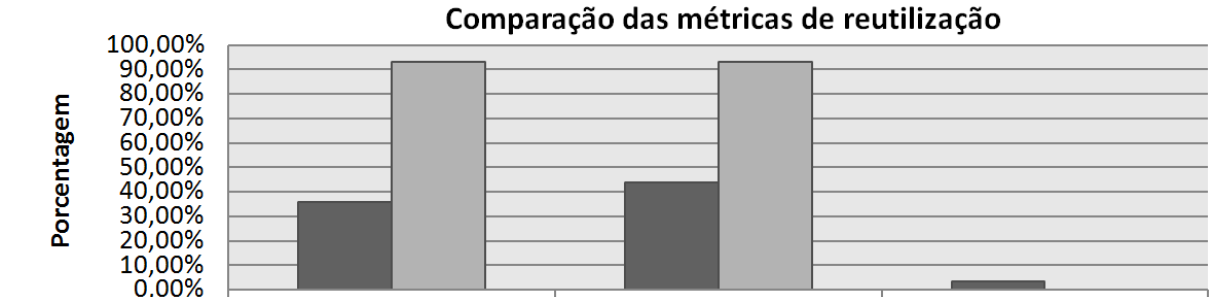
8.4 Resultados e discussão

8.4.1 Autoria de conteúdo para a Web

A Figura 28 mostra os valores das métricas de reutilização de software obtidas dos artefatos produzidos com e sem a abordagem. Nota-se um aumento significativo tanto na porcentagem de reutilização como na razão de reutilização. Também pode-se observar a redução da porcentagem de reutilização não desejada para zero.

Analisando-se os dados, verificou-se que a porcentagem de reutilização obtida exclusivamente com geração de código foi de **47,03%**, ou seja, quase metade do código reutilizado provém de um gerador. Foi verificada uma razão entre especificação e código de **1:16,34**, ou seja, para cada elemento de especificação são geradas, em média, pouco mais de 16 linhas de código.

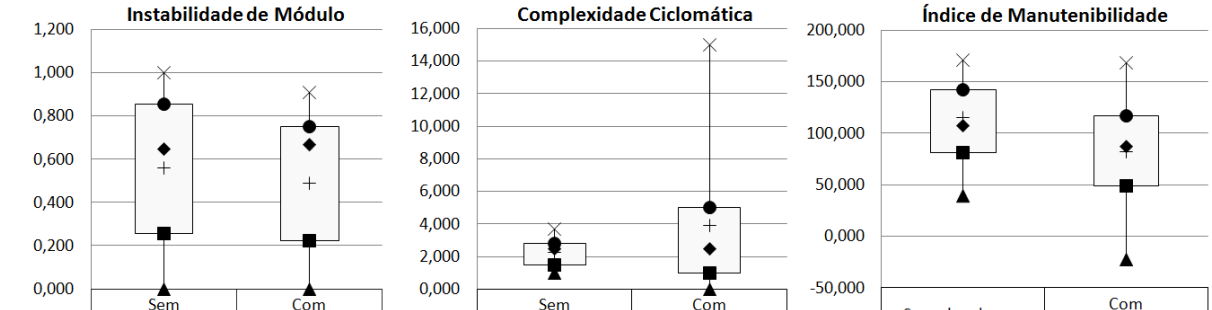
A Figura 29 mostra os valores das métricas indiretas de reusabilidade: instabilidade, complexidade e manutenibilidade obtidas dos artefatos produzidos com e sem a abordagem. Pode-se verificar que não houve alteração significativa nas distribuições da métrica de



	% de reutilização	Razão de reutilização	% de reutilização não desejada
Sem abordagem	36,04%	43,70%	3,32%
Com abordagem	93,32%	93,32%	0,00%

Figura 28: Comparação entre as métricas de reutilização para o estudo do domínio de autoria de conteúdo para a web

instabilidade. A complexidade, por sua vez, aumentou de forma perceptível, com muitos artefatos desenvolvidos com a abordagem sendo mais complexos do que o máximo obtido sem a abordagem. Os índices de manutenibilidade também sofreram uma redução com o uso da abordagem.



	Sem abordagem	Com abordagem
■q1	0,256	0,223
▲mín	0,000	0,000
◆mediana	0,646	0,667
+média	0,558	0,488
×máx	1,000	0,909
●q3	0,854	0,750

	Sem abordagem	Com abordagem
■q1	1,458	1,000
▲mín	1,000	0,000
◆mediana	2,500	2,500
+média	2,267	3,916
×máx	3,667	15,000
●q3	2,805	5,000

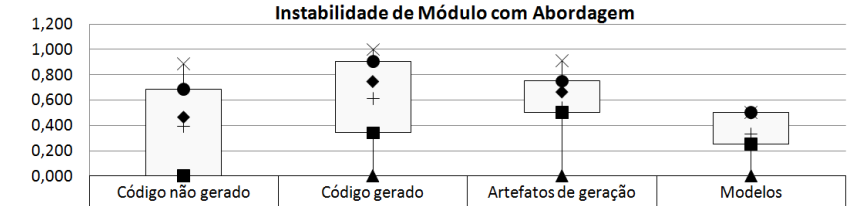
	Sem abordagem	Com abordagem
■q1	81,208	49,018
▲mín	38,910	-22,550
◆mediana	107,725	87,245
+média	114,891	82,329
×máx	171,000	168,560
●q3	142,155	116,493

Figura 29: Comparação entre as métricas indiretas de reusabilidade para o estudo do domínio de autoria de conteúdo para a web

A Figura 30 analisa de forma mais aprofundada as medidas de instabilidade obtidas com a abordagem[6]. Pode-se notar que o código gerado é mais instável do que o código não-gerado, em alguns casos chegando ao limite máximo desta métrica (1). Os artefatos de geração de código (transformações e geradores) apresentam uma distribuição intermediária às dos códigos

[6]Observação: neste gráfico, o código gerado foi incluído, apesar se ser um artefato não utilizado diretamente pelo desenvolvedor. Isto foi feito para ajudar na caracterização da reutilização apenas. O mesmo acontece em outros gráficos deste capítulo.

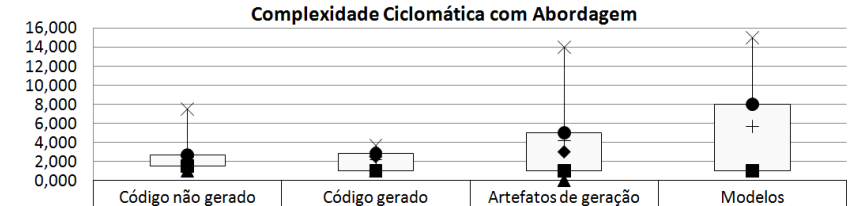
gerado e não-gerado, porém de forma mais concentrada em torno do valor central da métrica (0,5). Os modelos utilizados na abordagem tiveram sua instabilidade abaixo de 0,5, sendo em geral menos instáveis do que os demais artefatos.



	Código não gerado	Código gerado	Artefatos de geração	Modelos
■q1	0,000	0,341	0,500	0,250
▲mín	0,000	0,000	0,000	0,000
◆mediana	0,467	0,750	0,667	0,500
+média	0,393	0,614	0,537	0,333
×máx	0,889	1,000	0,909	0,500
●q3	0,683	0,907	0,750	0,500

Figura 30: Distribuições de instabilidade de módulo nos diferentes tipos de artefatos produzidos com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web

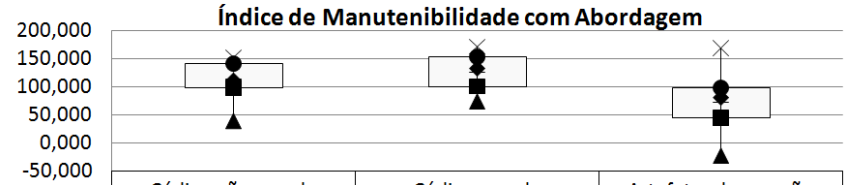
A Figura 31 analisa de forma mais aprofundada as medidas de complexidade ciclomática obtidas com a abordagem. Pode-se notar que o código gerado é semelhante, na média, ao código não-gerado, porém a distribuição do código não-gerado está um pouco mais concentrada. Os demais artefatos, incluindo transformações, geradores e modelos, são consideravelmente mais complexos do que o código, com alguns valores acima do limite considerado como o de módulos simples (10).



	Código não gerado	Código gerado	Artefatos de geração	Modelos
■q1	1,500	1,000	1,000	1,000
▲mín	1,000	1,000	0,000	1,000
◆mediana	2,500	2,500	3,000	1,000
+média	2,734	2,219	4,189	5,667
×máx	7,500	3,667	14,000	15,000
●q3	2,667	2,839	5,000	8,000

Figura 31: Distribuições de complexidade ciclomática nos diferentes tipos de artefatos produzidos com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web

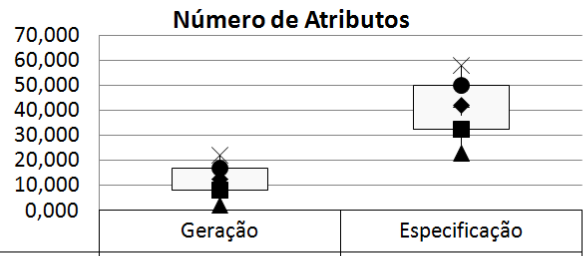
A Figura 32 analisa de forma mais aprofundada os índices de manutenibilidade obtidos com a abordagem. Pode-se notar que os elementos de geração de código (transformações e geradores) possuem menor índice de manutenibilidade do que o código, tanto gerado como não-gerado, que apresentaram distribuições próximas destas métricas. O código gerado, porém, apresenta valores ligeiramente maiores do índice de manutenibilidade.



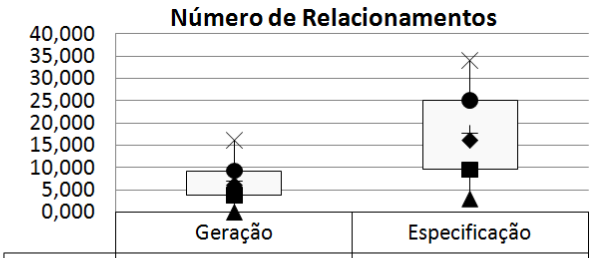
	Código não gerado	Código gerado	Artefatos de geração
■ q1	97,723	100,470	43,835
▲ mín	38,910	73,640	-22,550
◆ mediana	110,865	132,285	80,500
+ média	112,645	125,118	72,756
× máx	152,130	171,000	168,560
● q3	141,388	152,458	97,785

Figura 32: Distribuições do índice de manutenibilidade nos diferentes tipos de artefatos produzidos com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web

Para os modelos, a manutenibilidade foi analisada através do número de atributos e de relacionamentos. Conforme mostra a Figura 33, os modelos utilizados para geração de código (metamodelos e modelos auxiliares) possuem poucos atributos, permanecendo completamente abaixo do limite considerado (41). Também possuem poucos relacionamentos, com o terceiro quartil sendo similar ao limite considerado (9). Já os modelos utilizados como especificação possuem mais atributos e relacionamentos. A média do número de atributos coincide com o limite considerado (41), enquanto o número de relacionamentos tem distribuição bastante acima do limite considerado (9).



	Geração	Especificação
■ q1	8,000	32,500
▲ mín	2,000	23,000
◆ mediana	12,500	42,000
+ média	12,250	41,000
× máx	22,000	58,000
● q3	16,750	50,000



	Geração	Especificação
■ q1	3,750	9,500
▲ mín	0,000	3,000
◆ mediana	6,000	16,000
+ média	7,000	17,667
× máx	16,000	34,000
● q3	9,250	25,000

Figura 33: Distribuições do número de atributos e do número de relacionamentos nos modelos utilizados com a abordagem, no estudo de caso do domínio de autoria de conteúdo para web

Discussão

O primeiro e mais evidente resultado observado é o maior número de linhas de código reutilizadas com a abordagem, além de uma baixa taxa de reutilização não desejada. Uma

primeira explicação é a de que geradores produzem bastante código redundante, e por isso o número de linhas é naturalmente maior. Esta afirmação está próxima da realidade deste estudo, já que um dos itens importantes na identificação de candidatos a subdomínio é a existência de trechos repetidos de código, como ressaltado por Knodel et al. (2005) e discutido na atividade AD.3 desta abordagem (Capítulo 5). Neste estudo de caso, observa-se a existência de diversos arquivos e trechos de código parecidos sendo produzidos por geradores.

Vale ressaltar que esta repetição não poderia ser facilmente resolvida através de componentes ou outras estruturas de código, uma vez que apesar de serem basicamente repetidos, os trechos diferem em muitos pontos, e são resultados de inúmeros parâmetros e informações sobre o domínio. Como resultado, a tentativa de se implementar componentes genéricos que realizam estas variabilidades iria resultar em software mais complexo e difícil de ser mantido. A única opção, portanto, seria copiar os trechos parecidos e modificá-los manualmente, abordagem seguida no desenvolvimento tradicional. Em alguns casos, pode-se tentar aplicar técnicas de refatoração de código (FOWLER et al., 1999), mas estas são limitadas a somente alguns tipos de modificações no código.

Analisando os artefatos produzidos, nota-se que, na maioria deles, é exatamente esta tarefa de cópia e modificação manual que está sendo automatizada neste caso. A diferença entre os níveis de reutilização com e sem a abordagem (cerca de 50%) corresponde aproximadamente à porcentagem de reutilização obtida por geração (47,03%). Ou seja, o que está sendo gerado é o código que implementa a variabilidade que não pode ser automatizada em componentes, e foi construído manualmente sem a abordagem.

Outro dado que reforça este indício é o aumento da complexidade e a redução da manutenibilidade observados quando a abordagem é utilizada. Estas diferenças podem ser vistas de forma geral (Figura 29), mas são particularmente nítidas quando se avalia os diferentes tipos de artefatos do MDD individualmente (Figuras 30, 31 e 32). Os artefatos de geração de código e modelos de domínio são consideravelmente mais complexos e difíceis de manter do que o restante do código. Observando-se os artefatos, nota-se que isto se deve ao grande número de instruções condicionais, laços, e a existência de repetidas consultas aos modelos que são feitas nas transformações e geradores. Ou seja, o grande número de parâmetros citados anteriormente como necessários para realizar a variabilidade em um componente foram migrados para os artefatos do MDD, tornando-os complexos e difíceis de manter. No entanto, estes artefatos são modificados muito raramente, pois uma vez testados e validados, somente precisam ser alterados para efetuar correções de erros ou evolução no domínio.

Além disso, e esta é a principal diferença entre um gerador e um componente, um gerador

não é efetivamente reutilizado no software final, e sim o produto de sua geração de código, com características de reusabilidade próximas ao código não-gerado. Já os modelos, apesar de no geral serem mais complexos e difíceis de manter do que o código, são relativamente pequenos e em número reduzido, o que compensa sua complexidade.

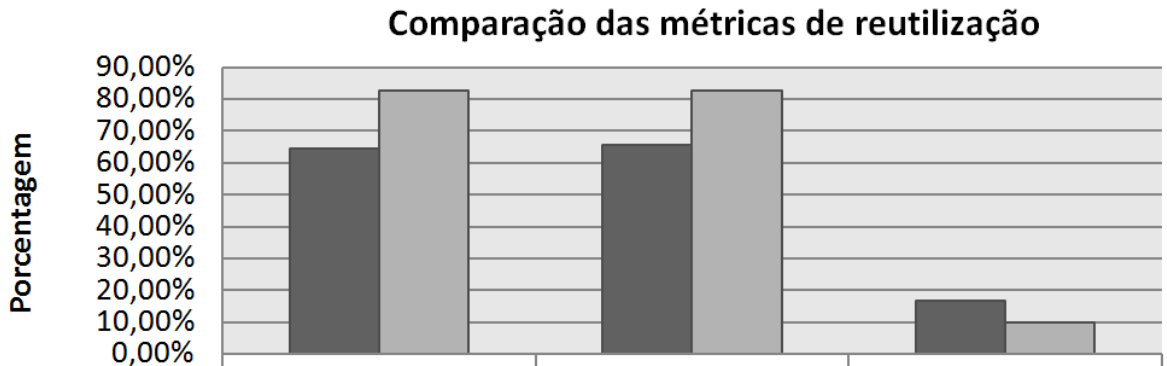
Enfim, o que se observa neste estudo é que ocorreu um aumento da reutilização, em detrimento da simplicidade e facilidade de manutenção de alguns de seus artefatos.

Em sistemas mais simples, pode não valer a pena o trabalho extra de se desenvolver uma infraestrutura deste tipo. Em outras palavras, é mais fácil e simples codificar do que especificar e gerar código. Porém, em sistemas grandes os benefícios do volume de reutilização podem ser muito grandes para serem ignorados. Considere por exemplo construir um sistema envolvendo milhares de telas de cadastros, todas parecidas e similares, muitas delas com detalhes e customizações necessárias. A complexidade extra neste caso pode ser um preço baixo a ser pago.

8.4.2 Aplicações distribuídas baseadas em computação em nuvem

A Figura 34 mostra os valores das métricas de reutilização de software obtidas dos artefatos produzidos com e sem a abordagem.

A exemplo do domínio web, porém em menor grau, observa-se um aumento significativo na porcentagem de reutilização e na razão de reutilização. Nota-se também que a porcentagem de reutilização não desejada caiu alguns pontos percentuais quando a abordagem foi utilizada.

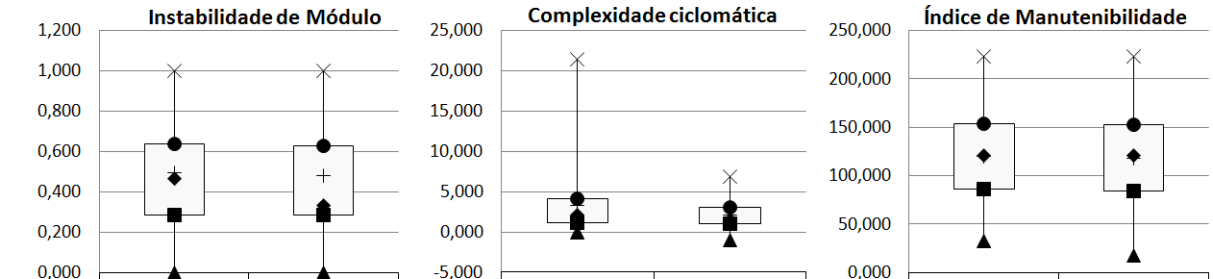


	% de reutilização	Razão de reutilização	% de reutilização não desejada
Sem abordagem	64,47%	65,66%	16,68%
Com abordagem	82,66%	82,66%	10,01%

Figura 34: Comparação entre as métricas de reutilização para o estudo do domínio de aplicações distribuídas baseadas em computação em nuvem

A porcentagem de reutilização obtida com geração de código neste estudo foi similar ao estudo anterior, atingindo **42,71%**. A razão entre especificação e código foi um pouco maior, atingindo **1:26,35**, ou seja, os geradores deste estudo produzem mais código para cada elemento de especificação do que os do estudo anterior.

A Figura 35 mostra os valores das métricas indiretas de reusabilidade: instabilidade, complexidade e manutenibilidade obtidas dos artefatos produzidos com e sem a abordagem. Pode-se verificar que não houve alteração significativa nas distribuições das métricas de instabilidade e índice de manutenibilidade. A complexidade, por sua vez, apresentou uma discreta redução.



	Sem abordagem	Com abordagem
■q1	0,286	0,286
▲mín	0,000	0,000
◆mediana	0,466	0,333
+média	0,496	0,479
×máx	1,000	1,000
●q3	0,636	0,625

	Sem abordagem	Com abordagem
■q1	1,121	1,000
▲mín	0,000	-1,000
◆mediana	2,100	1,449
+média	3,252	2,054
×máx	21,412	6,845
●q3	4,124	3,000

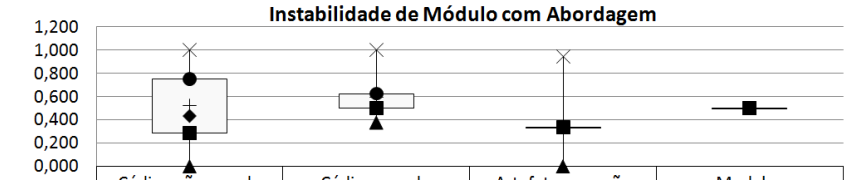
	Sem abordagem	Com abordagem
■q1	86,010	84,165
▲mín	32,510	17,210
◆mediana	121,230	121,230
+média	120,333	118,063
×máx	223,120	223,120
●q3	153,210	152,665

Figura 35: Comparação entre as métricas indiretas de reusabilidade para o estudo do domínio de aplicações distribuídas baseadas em computação em nuvem

A Figura 36 analisa de forma mais aprofundada as medidas de instabilidade obtidas com a abordagem. Pode-se notar que o código gerado é similar, na média, ao código não-gerado. Porém, a distribuição referente ao código gerado está mais concentrada ao redor do valor médio desta métrica (0,5). Os artefatos de geração, por sua vez, estão concentrados em um ponto que representa baixa instabilidade. Com relação aos modelos, neste caso somente um modelo foi utilizado, e portanto a métrica de instabilidade permaneceu no valor intermediário (0,5).

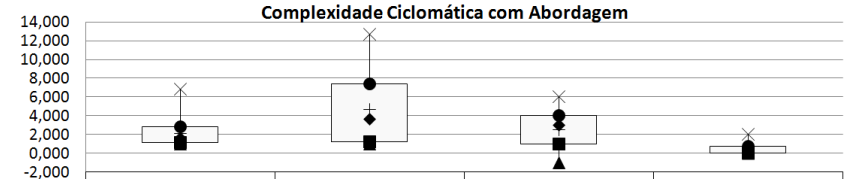
A Figura 37 analisa de forma mais aprofundada as medidas de complexidade ciclomática obtidas com a abordagem. Pode-se notar que o código gerado apresenta maior complexidade em relação ao código não-gerado. Os artefatos de geração são ligeiramente mais complexos do que o código não-gerado, e os modelos apresentam complexidade baixa. Com exceção do código gerado, todos os artefatos permaneceram dentro do limite que indica simplicidade (10).

A Figura 38 analisa de forma mais aprofundada os índices de manutenibilidade obtidos com a abordagem. Não existem diferenças notáveis na distribuição, porém há uma leve queda



	Código não gerado	Código gerado	Artefatos geração	Modelos
■q1	0,286	0,500	0,333	0,500
▲mín	0,000	0,375	0,000	0,500
◆mediana	0,432	0,500	0,333	0,500
+média	0,522	0,590	0,373	0,500
×máx	1,000	1,000	0,944	0,500
●q3	0,750	0,625	0,333	0,500

Figura 36: Distribuições de instabilidade de módulo nos diferentes tipos de artefatos produzidos com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem



	Código não gerado	Código gerado	Artefatos geração	Modelos
■q1	1,113	1,200	1,000	0,000
▲mín	1,000	1,000	-1,000	0,000
◆mediana	1,600	3,644	3,000	0,000
+média	2,128	4,673	2,524	0,357
×máx	6,845	12,714	6,000	2,000
●q3	2,800	7,420	4,000	0,750

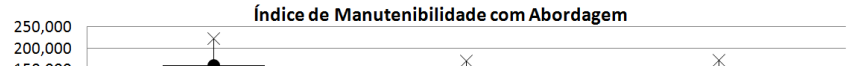
Figura 37: Distribuições de complexidade ciclomática nos diferentes tipos de artefatos produzidos com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem

na manutenibilidade do código gerado e nos artefatos de geração, com relação ao código gerado.

A Figura 39 mostra que as quantidades de atributos de ambos os tipos de modelo (geração e especificação) permaneceram abaixo do limite estipulado (41). O número de relacionamentos, porém, excedeu consideravelmente o limite (9) nos modelos de geração, e de forma moderada no caso dos modelos de especificação.

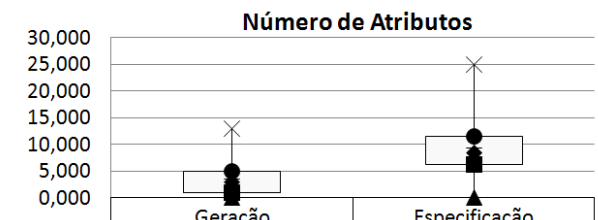
Discussão

Neste domínio, a exemplo do domínio Web mas de forma menos acentuada, também foram observados aumentos no nível e na razão de reutilização de linhas de código, e uma baixa taxa de reutilização não desejada. Os mesmos motivos discutidos no estudo do domínio Web se aplicam a este caso, sendo observados diversos artefatos com alto grau de similaridade. Esta parece ser uma tendência de domínios técnicos, onde a tônica está em resolver problemas de mais baixo nível de forma repetida e constante, como a persistência e navegação, no caso do domínio Web, e distribuição, no caso deste domínio. Neste caso, inclusive, foi observada uma

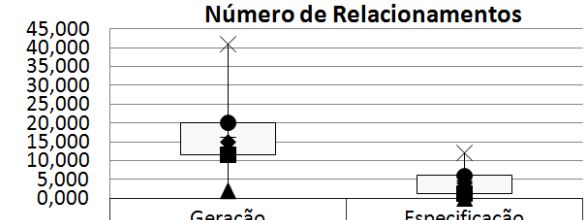


	Código não gerado	Código gerado	Artefatos geração
■q1	91,515	70,750	77,210
▲mín	32,510	33,260	17,210
◆mediana	133,355	106,250	89,510
+média	127,499	100,101	96,589
×máx	223,120	171,000	172,050
●q3	160,908	121,140	125,510

Figura 38: Distribuições do índice de manutenibilidade nos diferentes tipos de artefatos produzidos com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem



	Geração	Especificação
■q1	1,000	6,250
▲mín	0,000	0,000
◆mediana	3,000	8,500
+média	3,500	9,357
×máx	13,000	25,000
●q3	5,000	11,500



	Geração	Especificação
■q1	11,500	1,250
▲mín	2,000	0,000
◆mediana	15,000	4,000
+média	16,188	4,643
×máx	41,000	12,000
●q3	20,000	6,000

Figura 39: Distribuições de números de atributos e relacionamentos nos modelos utilizados com a abordagem, no domínio de aplicações distribuídas baseadas em computação em nuvem

razão ainda maior entre especificação e código (1:26,35), o que indica que os geradores estão sendo mais produtivos.

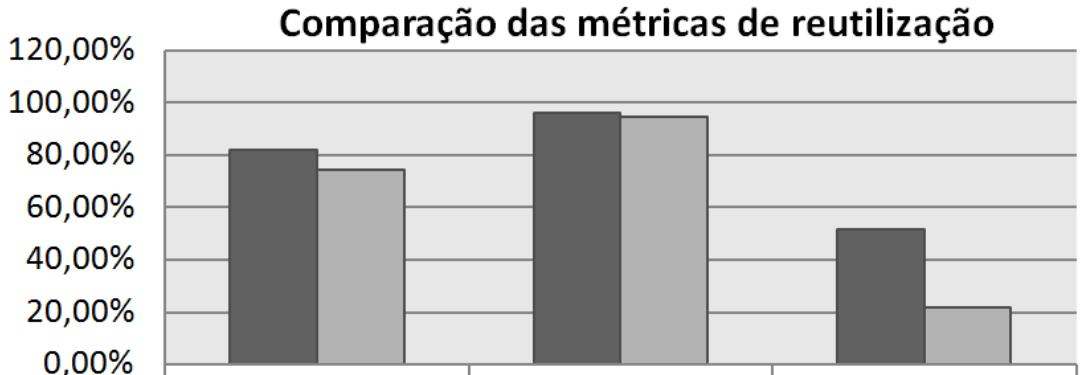
Porém, enquanto no estudo anterior os artefatos de geração de código e os modelos são mais complexos do que o código, aqui foi observado o efeito inverso. O código, tanto o gerado como o não-gerado, é mais instável, complexo e difícil de manter do que os artefatos de geração. Isto se explica pela complexidade natural do domínio, uma vez que a computação em nuvem traz muitos desafios não triviais, como o cálculo distribuído de invariantes do sistema, determinação de cliques maximais, descoberta dinâmica de serviços e execução distribuída (LUCRÉDIO; JACKSON; SCHULTE, 2008). Detalhes sobre estes assuntos ficam encapsulados não somente em componentes, mas também nos geradores, de forma que um desenvolvedor pode produzir aplicações complexas mesmo sem conhecer a fundo todas as tecnologias envolvidas. Em outras palavras, em contraste com o estudo anterior, neste domínio é mais fácil e simples especificar do que codificar.

Assim, o que se observou neste estudo foi, além do aumento na reutilização, um aumento na reusabilidade dos artefatos do domínio, percebido de forma indireta com a redução da instabilidade, complexidade e dificuldade de manutenção.

Assim, enquanto no estudo anterior verificou-se que o uso da abordagem pode trazer benefícios em termos de volume de reutilização, aqui observa-se que a abordagem pode simplificar o desenvolvimento, abstraindo detalhes e complexidades inerentes a este domínio complexo.

8.4.3 Eventos científicos

A Figura 40 ilustra uma comparação entre os níveis de reutilização obtidos com e sem a abordagem, para o estudo envolvendo o domínio de eventos científicos. Nota-se que, ao contrário dos estudos anteriores, com a abordagem tem-se menores taxas e razões de reutilização. Porém, a taxa de reutilização não desejada foi reduzida para menos da metade. Além disso, neste domínio nota-se uma maior diferença entre a porcentagem de reutilização e a razão de reutilização, do que nos domínios anteriores. Isto porque há um grande número de artefatos que são reutilizados e modificados minimamente (reutilização do tipo *caixa-branca*), como resultado do processo de customização.



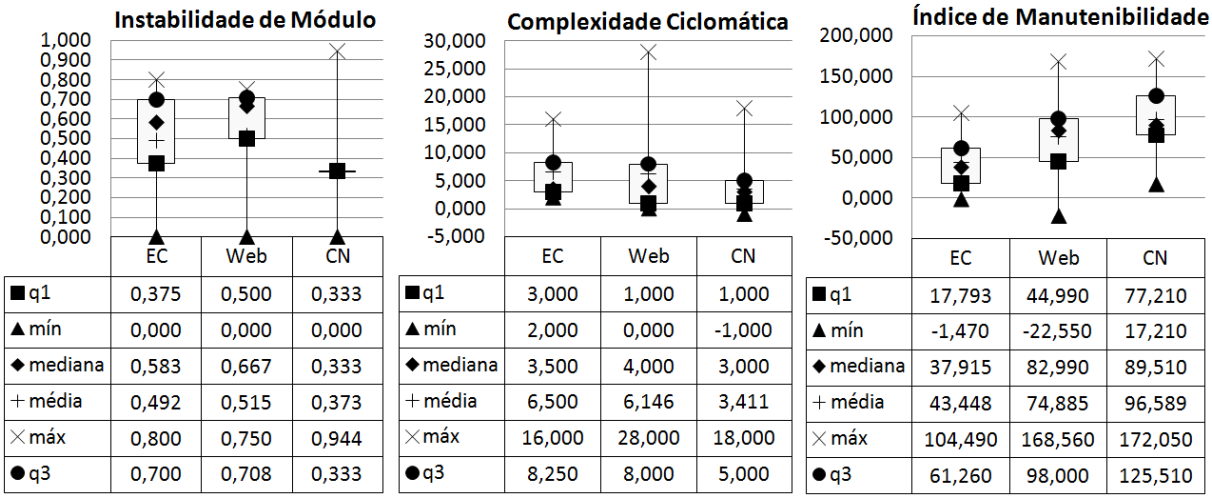
	% de reutilização	Razão de reutilização	% de reutilização não desejada
Sem abordagem	82,15%	96,11%	51,43%
Com abordagem	74,45%	94,69%	21,88%

Figura 40: Comparação entre as métricas de reutilização para o estudo do domínio de eventos científicos

A porcentagem de reutilização obtida com geração de código neste estudo foi bastante reduzida, sendo medida como **3,99%**. A razão entre especificação e código também foi significativamente menor, atingindo **1:8,14**, ou seja, os geradores deste estudo produzem menos

código para cada elemento de especificação do que os dos estudos anteriores.

Conforme discutido anteriormente, neste estudo não foram coletadas métricas para os artefatos de código, por motivos de falta de ferramentas adequadas. Porém, foram analisados os atributos dos artefatos de geração, para efeitos de comparação com os outros estudos. Esta comparação é apresentada na Figura 41. Pode-se notar que, em termos de instabilidade, os artefatos deste estudo estão mais próximos aos do estudo envolvendo o domínio Web, porém ligeiramente menos instáveis. Em termos de complexidade ciclomática, a distribuição ficou mais concentrada e ligeiramente superior que os outros estudos. O índice de manutenibilidade se mostrou notavelmente inferior aos outros estudos.



	EC	Web	CN
■ q1	0,375	0,500	0,333
▲ mín	0,000	0,000	0,000
◆ mediana	0,583	0,667	0,333
+ média	0,492	0,515	0,373
× máx	0,800	0,750	0,944
● q3	0,700	0,708	0,333

	EC	Web	CN
■ q1	3,000	1,000	1,000
▲ mín	2,000	0,000	-1,000
◆ mediana	3,500	4,000	3,000
+ média	6,500	6,146	3,411
× máx	16,000	28,000	18,000
● q3	8,250	8,000	5,000

	EC	Web	CN
■ q1	17,793	44,990	77,210
▲ mín	-1,470	-22,550	17,210
◆ mediana	37,915	82,990	89,510
+ média	43,448	74,885	96,589
× máx	104,490	168,560	172,050
● q3	61,260	98,000	125,510

Figura 41: Comparação entre as métricas de reusabilidade dos estudos dos domínios de eventos científicos (EC), Web e Computação em Nuvem (CN)

Foram utilizados somente dois modelos baseados em XML, e estes apresentaram uma média de 36,33 atributos e nenhum relacionamento.

Com relação aos dados qualitativos, foram coletadas algumas impressões passadas pela equipe após a execução do estudo. Entre as impressões obtidas com a entrevista[7], destacam-se:

- Segundo a equipe, a abordagem permitiu atacar problemas recorrentemente enfrentados pela equipe, tais como o alto nível de retrabalho procurando códigos e testando cada mudança no domínio, a existência de arquivos que nunca são utilizados mas que são incluídos nos produtos por conveniência, o que acaba por confundir os desenvolvedores e causando problemas de manutenção, e a necessidade de busca por *links* que precisam removidos dependendo da configuração de produtos adquiridos pelo cliente;

- Neste estudo, a automação conseguida com o MDD permitiu reduzir alguns problemas de

[7] O conteúdo integral da entrevista encontra-se no Apêndice C.

forma que não vinha sendo conseguida, por falta de tempo e dificuldades em desenvolver uma solução que atendesse a múltiplos clientes ao mesmo tempo;

- A equipe relatou dificuldades no aprendizado das tecnologias de modelagem e geração de código, porém com relação à abordagem citaram apenas que tiveram dificuldades em compreender as funções específicas dos casos de mudança na abordagem. Segundo eles, não ficou clara a necessidade de se criar múltiplos cenários para descrever pequenas partes do problema;

- A equipe teve também dificuldades na identificação correta das *features*, sendo necessária a presença constante do autor desta tese para orientar e corrigir as identificações equivocadas. No geral, os participantes compreenderam os conceitos, mas tinham dificuldade em reproduzi-los no domínio em questão, inserindo constantemente e de maneira equivocada, módulos e funções como sendo *features*; e

- A equipe não soube responder de forma apropriada com relação ao processo investigativo de identificação de subdomínios, à atividade de avaliação arquitetural, e à atividade de documentação do domínio, pois não chegou a realizar estas atividades durante o estudo.

Discussão

Neste estudo, ao contrário dos dois anteriores, ocorreu uma redução da porcentagem e da razão de reutilização, quando a abordagem foi utilizada. Porém, conforme discutido anteriormente, a métrica de reutilização em termos de LOC não pode ser lida literalmente, de forma isolada, pois pode ser distorcida por alguns fatores. Conforme a própria equipe relatou em entrevista, há muitos artefatos que são reutilizados desnecessariamente. Além de causar prejuízos à manutenção, ocupar espaço de forma desnecessária e confundir os desenvolvedores, este fato distorce a métrica de reutilização.

Analisando-se também a quantidade de reutilização não desejada, nota-se que houve redução significativa, ou seja, na realidade a quantidade de LOC reutilizadas foi menor, porém a reutilização ocorreu de forma melhor, com menos código desnecessário poluindo a aplicação. Isto foi alcançado de forma simples, através de geradores que fazem a cópia seletiva dos arquivos de acordo com as configurações do produto.

Os artefatos produzidos pela equipe apresentaram qualidade inferior, em termos de instabilidade, complexidade e manutenibilidade, do que os artefatos desenvolvidos nos outros estudos. Isto se deve principalmente às dificuldades com o aprendizado e treinamento, conforme citado pela própria equipe. Além disso, foram desenvolvidos poucos artefatos e uma baixa taxa

de reutilização com geração de código (3,99%), devido principalmente à falta de tempo e à dedicação parcial dos participantes. Estes resultados são particularmente interessantes, pois representam a utilização da abordagem em um cenário mais típico e próximo da realidade de uma organização de software.

Existe também uma ocorrência que foi brevemente citada pela equipe e que merece uma avaliação mais aprofundada. Na entrevista, os participantes citaram que com a abordagem puderam resolver problemas que não conseguiam resolver anteriormente, não somente pela falta de tempo, mas também pela dificuldade em se obter soluções genéricas que possam ser utilizadas por múltiplos clientes. Em muitos casos, as modificações exigidas por um cliente não podem ser generalizadas e parametrizadas, pois a natureza das mudanças é profunda, como por exemplo as diferentes formas de certificado, citadas como exemplo pela equipe. É possível criar um componente genérico capaz de gerar certificados para eventos com diferentes textos, tamanhos de fontes, de papel, entre outros parâmetros. Porém, é comum a existência de chamados solicitando a presença de informações extras, como um determinado texto em negrito, orientações de texto diferentes, e dados não-convencionais, como por exemplo certificados de responsáveis por sessões (*chair* de sessão). Nestes casos, a única solução encontrada era criar uma cópia do componente anterior e modificá-la para o novo requisito. Com o MDD e a abordagem, a equipe conseguiu criar um suporte mais flexível para estas modificações, sem perder a generalidade do componente original e sem causar a duplicação de versões.

Outro problema citado é o espalhamento da informação, o que exige um grande trabalho de inspeção, modificação e testes a cada configuração de um novo produto. Este espalhamento é visível nos dados coletados: no estudo realizado, em torno de 70 artefatos foram modificados em menos de 25%. Estas modificações envolvem configurações e customizações, muitas delas duplicadas. Com a abordagem, foi possível reunir grande parte dos parâmetros de configuração em um único modelo, e as modificações pontuais e repetitivas são realizadas por geradores. Neste estudo, o modelo serve de entrada para configurações em seis tabelas diferentes do banco, e quatro arquivos de configuração, que antes precisavam ser feitas individualmente. Além disso, os geradores são mais confiáveis nesta tarefa, reduzindo a necessidade de testes mais exaustivos.

As métricas de número de atributos e relacionamentos para os modelos utilizados neste estudo mostram que existem muitos atributos (média de 36,33) e nenhum relacionamento, o que indica que se tratam de modelos que servem basicamente de configuração apenas.

8.5 Análise das hipóteses e conclusões

Os dados e resultados obtidos com os estudos possuem indícios sobre a rejeição de algumas das hipóteses nulas. Esta rejeição é feita com base em análise descritiva apenas, não sendo embasada por dados estatísticos rigorosos.

Com relação à hipótese H_{0a}, conclui-se que ela é rejeitada, pois nos três estudos foram verificados aumento e/ou melhoria na reutilização de software nos projetos utilizando a abordagem.

Com relação à hipótese H_{0b}, não foi possível alcançar a rejeição, uma vez que os resultados indicaram que a reusabilidade dos artefatos pode ser maior ou menor utilizando-se a abordagem.

Com relação às hipóteses H_{0c} e H_{0d}, os dados obtidos possuem indícios que apóiam a rejeição, uma vez que os participantes dos estudos perceberam benefícios com a abordagem, e não tiveram dificuldades significativas no aprendizado.

Mesmo para as hipóteses nulas com indícios de rejeição, não se pode afirmar que as hipóteses alternativas apresentadas na Seção 8.1.2 são verdadeiras. Ou seja, a abordagem aparentemente favorece a reutilização, mas pode ser que existam projetos onde ela não acrescente benefícios ou mesmo cause dificuldades significativas de aprendizado e utilização.

O que parece ser uma conclusão mais razoável é que a abordagem pode aumentar a quantidade, em termos de LOC. Mas ela não aumenta, de forma absoluta, os valores das métricas de complexidade e dificuldade de manutenção, conforme percebido no estudo do domínio Web. Tampouco ela as reduz, conforme percebido no estudo do domínio de computação em nuvem. Aparentemente, há uma tendência a produzir artefatos com instabilidade, complexidade e dificuldade de manutenção acima da média, mas seguindo uma constante, e dependendo das características do domínio, isto pode ser vantajoso ou não. Este efeito é ilustrado na Figura 42.

Assim, uma possível hipótese alternativa mais realista deve considerar as diferentes condições e cenários dos domínios, conforme ilustra a Figura 43. Em domínios técnicos mais simples, a abordagem proporciona maior quantidade de reutilização mas menor reusabilidade dos artefatos, sendo mais indicada para quando há a necessidade de geração de grande volume de código. Em domínios técnicos mais complexos, a abordagem simplifica o desenvolvimento quando a codificação é extremamente complexa. Em domínios de negócio, ela pode facilitar o processo de configuração e reduzir os problemas causados com a proliferação de versões.

Esta hipótese alternativa resume muitas das experiências vivenciadas nesses três estudos,

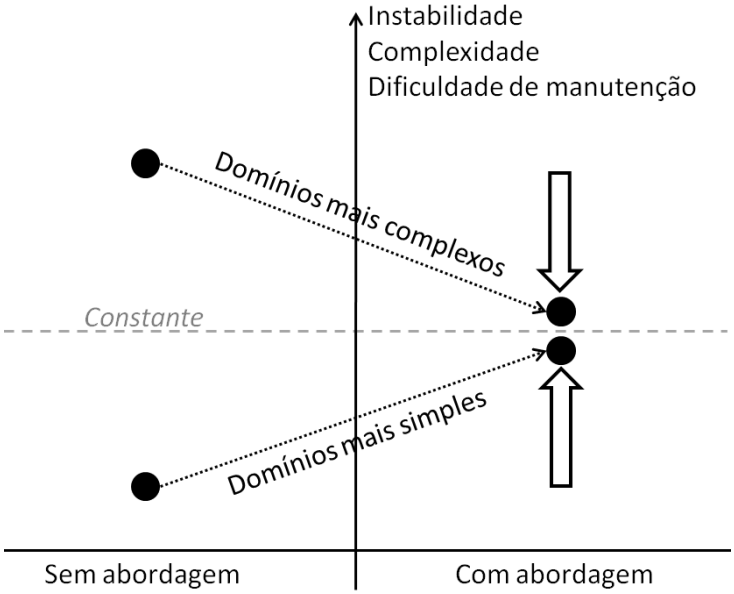


Figura 42: Efeito da abordagem na reusabilidade dos artefatos produzidos

e explica de forma plausível os efeitos observados. Porém, ela foi extrapolada, não sendo embasada por evidências mais sólidas. Estudos mais rigorosos podem e devem ser realizados para complementar estes resultados e o conhecimento adquiridos nesta tese, para que generalizações estatisticamente comprovadas possam ser formuladas.

8.6 Ameaças à validade

Os dados obtidos com esta avaliação são bastante informativos e abrangentes. Porém, há muitas questões que sequer chegaram a ser exploradas, tais como o impacto da abordagem nos níveis de reutilização internos, como aqueles alcançados com herança, por exemplo. Além disso, algumas etapas da abordagem não puderam ser avaliadas, como a atividade de avaliação arquitetural e a documentação dos artefatos.

Deve-se citar que a participação do autor da tese em dois dos estudos pode ter influenciado negativamente os resultados, sendo uma ameaça à sua validade, uma vez que tendo ele próprio desenvolvido a abordagem, não passou por dificuldades em seu aprendizado, e provavelmente falhou em identificar pontos fracos da mesma, dois aspectos importantes da avaliação. Ainda assim considerou-se que os resultados obtidos são válidos, pois muitos deles foram obtidos a partir de métricas objetivas, que não sofrem a influência do pesquisador. As métricas subjetivas foram coletadas diretamente dos participantes do terceiro estudo, e portanto também não foram influenciadas.

Vale mencionar também o fato de que a realização dos estudos em ambiente industrial

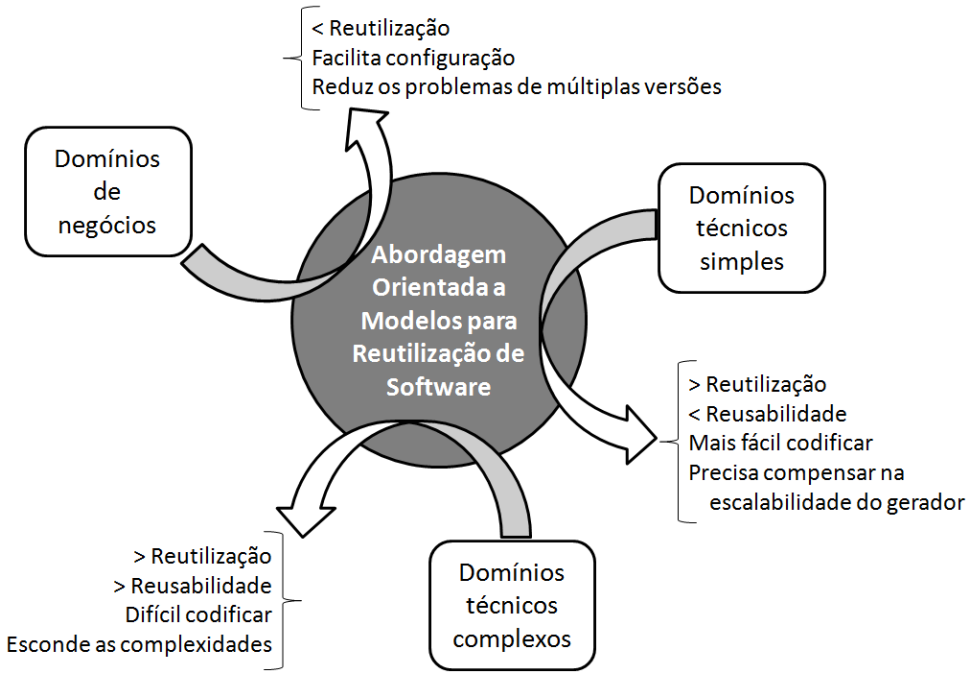


Figura 43: Uma possível hipótese alternativa elaborada a partir dos resultados da avaliação

apresenta um menor grau de controle, uma vez que o desenvolvimento sofre influências e pressões constantes, naturais deste tipo de ambiente. Como resultado, a qualidade e rigor dos dados é menor do que em um estudo controlado. Por ser esta uma tese na área de Engenharia de Software, não se pode descartar este tipo de estudo, afinal, a indústria deve ser o destino imediato ou a longo prazo de toda e qualquer pesquisa relevante nesta área. As lições aprendidas, ainda que em forma de comentários e impressões subjetivas, devem ser consideradas como sendo relevantes e valiosas.

O tamanho dos projetos envolvidos nos estudos, que se caracterizam entre pequeno e médio portes, é razoável para este tipo de avaliação, e não foi considerado como uma ameaça à validade. Dado o tamanho das equipes ser muito reduzido, não foi possível avaliar aspectos referentes ao trabalho colaborativo, nem a capacidade da abordagem em atender aos requisitos deste tipo de desenvolvimento.

Devido também ao tamanho dos projetos, pode-se citar o fato de que a análise de dispersão e balanceamento das amostras permitiu identificar alguns poucos elementos fora da distribuição normal. Estes, porém, pouco influenciaram os resultados. Em futuras replicações deste experimento, pode-se optar por ignorar esta análise sem muitos prejuízos aos resultados, caso os tamanhos dos projetos sejam similares aos aqui apresentados.

A comparação de implementações obtidas com e sem a abordagem pode também ter sofrido influência pelo fato de que a abordagem foi utilizada após o desenvolvimento de uma implementação de exemplo. Nos dois primeiros estudos, essa implementação consistiu de

protótipos executáveis desenvolvidos para praticar o desenvolvimento no domínio, e no terceiro estudo, consistiu de um produto real existente na empresa. Com isso, o desenvolvimento com a abordagem pode ter se beneficiado de uma maior experiência no domínio. Porém, a própria abordagem utiliza uma estratégia de desenvolvimento através de exemplos para proporcionar justamente este ganho de experiência antes da implementação efetiva dos artefatos do MDD e, portanto, este efeito é parcialmente o que seria obtido com a abordagem de qualquer forma. Mas uma avaliação mais justa deveria envolver equipes diferentes partindo de um mesmo nível de conhecimento sobre o domínio. Isto não foi possível realizar pela dificuldade em encontrar participantes para estes estudos.

Finalmente, destacam-se as ameaças às validades interna, externa e de construção (WOHLIN et al., 2000). Os estudos foram planejados, definidos e executados de forma mais exploratória, sem a definição de variáveis dependentes e independentes voltadas a uma análise de correlação mais rígida. Por exemplo, os valores de referência e margens de tolerância citados servem mais como guia, e não como um indicador efetivo do efeito observado. Isto prejudica a sua capacidade de replicação, tanto no mesmo grupo como em grupos de pesquisa externos, mas também prejudica a relação causa-efeito, uma vez que não se pode afirmar que os efeitos observados são devidos à utilização da abordagem ou ocorreram por outros fatores não previstos. Assim, é necessário um maior detalhamento das variáveis e aspectos envolvidos para transformar estes estudos em um experimento de caráter exclusivamente científico.

Também não foi realizada nenhuma avaliação direta do processo. Somente foram analisados os produtos (artefatos de software) desenvolvidos. Apesar destes sofrerem influência direta do processo, não se pode afirmar que os resultados observados são efetivamente causados pela abordagem, sem uma análise mais dedicada de suas atividades e da execução realizada.

Ressalta-se também o fato de que o processo de identificação de código reutilizado mas não referenciado, descrito na Seção 8.1.1, pode levar a falsos negativos. Por exemplo, podem existir métodos sendo chamados dentro de blocos condicionais que nunca são acessados.

8.7 Considerações finais

Neste capítulo foi apresentada uma avaliação da tese sendo defendida nesta dissertação. Foram realizados três estudos empíricos, um em ambiente acadêmico e dois em ambiente industrial, visando caracterizar os efeitos da utilização da abordagem na reutilização de software. Com exceção das ameaças à validade citadas na seção anterior, os estudos contêm resultados e indícios que permitiram a análise de algumas conclusões relevantes ao contexto

desta tese.

A avaliação permitiu determinar os principais benefícios, mas também algumas falhas da abordagem, como por exemplo o fato de que os participantes do terceiro estudo não perceberam utilidade na identificação de cenários para descrever as variabilidades nos comportamentos do domínio, alegando ser esta uma tarefa desnecessária. Esta limitação indica que talvez seja preciso uma reestruturação da abordagem, visando aumentar sua agilidade, ou mesmo um melhor treinamento visando reforçar a importância desta atividade. A escolha de qual direção a ser tomada depende de uma análise mais aprofundada do problema.

Também foram identificadas falhas na própria avaliação, principalmente no terceiro estudo, como por exemplo a não utilização do formato de documentação sugerido e nem dos processos investigativos para identificação dos subdomínios e de análise arquitetural. Estas são decorrentes principalmente de restrições à execução do estudo impostas pelo próprio cenário industrial em que foi realizado.

Vale a pena ressaltar que os estudos foram realizados em diferentes ambientes, utilizando diferentes tecnologias para o MDD, como EMF, GMF, *openArchitectureWare*, *Microsoft Text Templates*, GME, e em diferentes plataformas e linguagens de programação, como Java, C# e PHP. Isto reforça a validade dos resultados e a sua independência com relação às tecnologias envolvidas.

9 *Trabalhos relacionados*

As idéias do MDD estão fortemente ligadas com o uso de ferramentas de auxílio ao desenvolvimento de software. Por esse motivo, não é estranho o fato de que a indústria esteja voltando seus olhos para esse paradigma, uma vez que fabricantes de ferramentas vêem um forte indício de vantagem competitiva caso consigam oferecer a seus clientes uma maneira de alcançar os benefícios de qualidade e produtividade associados a esse paradigma de desenvolvimento. Na Seção 2.2.2 foram apresentadas as principais abordagens da indústria para o MDD.

Mas com a academia não é diferente, sempre existindo o interesse científico nessa área, com trabalhos que discutem os conceitos teóricos e a viabilidade dessa abordagem.

9.1 Abordagens orientadas a modelos para reutilização de software

Um dos primeiros trabalhos a propor a uma abordagem similar ao que se busca hoje com MDD data de 1980 (NEIGHBORS, 1980). Pioneiro na área de reutilização de software, James Neighbors, com sua abordagem Draco para desenvolvimento de software, também investigou a viabilidade de se utilizar modelos como a base para a construção de aplicativos.

Nas palavras de Neighbors, a abordagem Draco se baseia na "*sensação frustrante de que a maior parte do sistema que você está construindo atualmente é a mesma que você já construiu em alguns sistemas anteriores*" (NEIGHBORS, 1989). Sua proposta é que seja feita uma análise sobre um domínio de aplicações similares, a exemplo do que propôs Parnas (1976), e que o conhecimento obtido seja representado por meio de linguagens específicas para este domínio (DSL) e para os domínios relacionados, de forma a facilitar sua reutilização ao se construir um novo sistema. Essa proposta é muito similar às idéias do MDD.

Na abordagem Draco, um analista do domínio, normalmente uma pessoa com experiência na construção de diferentes aplicações similares, cria a descrição desse domínio segundo uma

DSL. Um domínio no Draco inclui um interpretador semântico (*parser*), que incorpora as regras gramaticais da linguagem. Desta forma, ao se construir um sistema para esse domínio, um engenheiro de software pode utilizar essa DSL.

Por exemplo, considere um analista do domínio de jogos eletrônicos. Ele define elementos como “placar”, “comando”, “fase”, “jogador”, entre outros, e os descreve utilizando regras gramaticais, que servem para a criação de um interpretador (*parser*). O analista de um sistema utiliza esses elementos para construir seu próprio jogo, criando elementos como “Jogador 1”, “Jogador 2”, “comandoDireita”, “comandoEsquerda”, etc.

Uma possível gramática para essa DSL, onde um jogo envolve vários jogadores, seria:

```
jogo      : 'Jogo' nome ':' jogadores 'Fim';
jogadores : (jogador)*;
jogador   : 'Jogador' nome 'Cor' cor;
cor       : 'verde'|'azul'|'branco'|'preto';
nome      : '"' (a-zA-z)(a-zA-z0-9)* '"';
```

Dessa forma, um novo jogo poderia ser definido por meio dessa linguagem, por exemplo:

```
Jogo "Damas" :
   Jogador "J1" Cor branco
   Jogador "J2" Cor preto
Fim
```

No Draco, o analista do domínio também pode construir transformações que permitem que os elementos criados pelo analista do sistema sejam automaticamente transformados em domínios intermediários, até eventualmente chegar em linguagem executável.

É o que acontece quando, por exemplo, se deseja obter um modelo orientado a objetos deste jogo, para posteriormente implementá-lo utilizando uma linguagem de programação orientada a objetos. No Draco, as transformações se baseiam nas regras gramaticais das linguagem dos domínios, interpretadas pelo seu próprio *parser*. Por isso é necessário existir uma gramática para a linguagem de origem e uma gramática para a linguagem destino. Neste caso, seria necessária uma gramática para esse modelo orientado a objetos intermediário, que poderia ser parecida com:

```
modelo     : 'modelo' '{' (classe)* (objeto)* '}';
```

```
classe     : 'classe' nome '{' (atributo)* '}';
atributo   : nome ':' tipo;
tipo       : 'String'|'int'|'float'|nome;
nome       : (a-zA-z)(a-zA-z0-9)*;
objeto     : 'objeto' nome ':' nome '(' ((parametro)(',' parametro)*)? ')';
parametro  : '"' (a-zA-z0-9)* '"';
```

Uma transformação automática deste jogo para este modelo orientado a objetos poderia gerar algo como:

```
modelo {
   classe Jogador {
      nome: String
      cor: String
   }

   objeto objJ1:Jogador("J1","branco")
   objeto objJ2:Jogador("J2","preto")
}
```

De forma similar, com base na definição do domínio para uma linguagem executável, como por exemplo Java, poderia ser definida uma transformação que gerasse o seguinte código para este jogo:

```
public class Jogador {
   private String nome;
   private String cor;

   public Jogador(String nome, String cor) {
      this.nome = nome;
      this.cor = cor;
   }
}

public class Main {
   public static void main (String args[]) {
```

```
        Jogador objJ1 = new Jogador("J1","branco");
        Jogador objJ2 = new Jogador("J1","preto");
    }
}
```

Esse processo é chamado no Draco de "*ciclo de refinamento básico*", e é resumido na Figura 44.

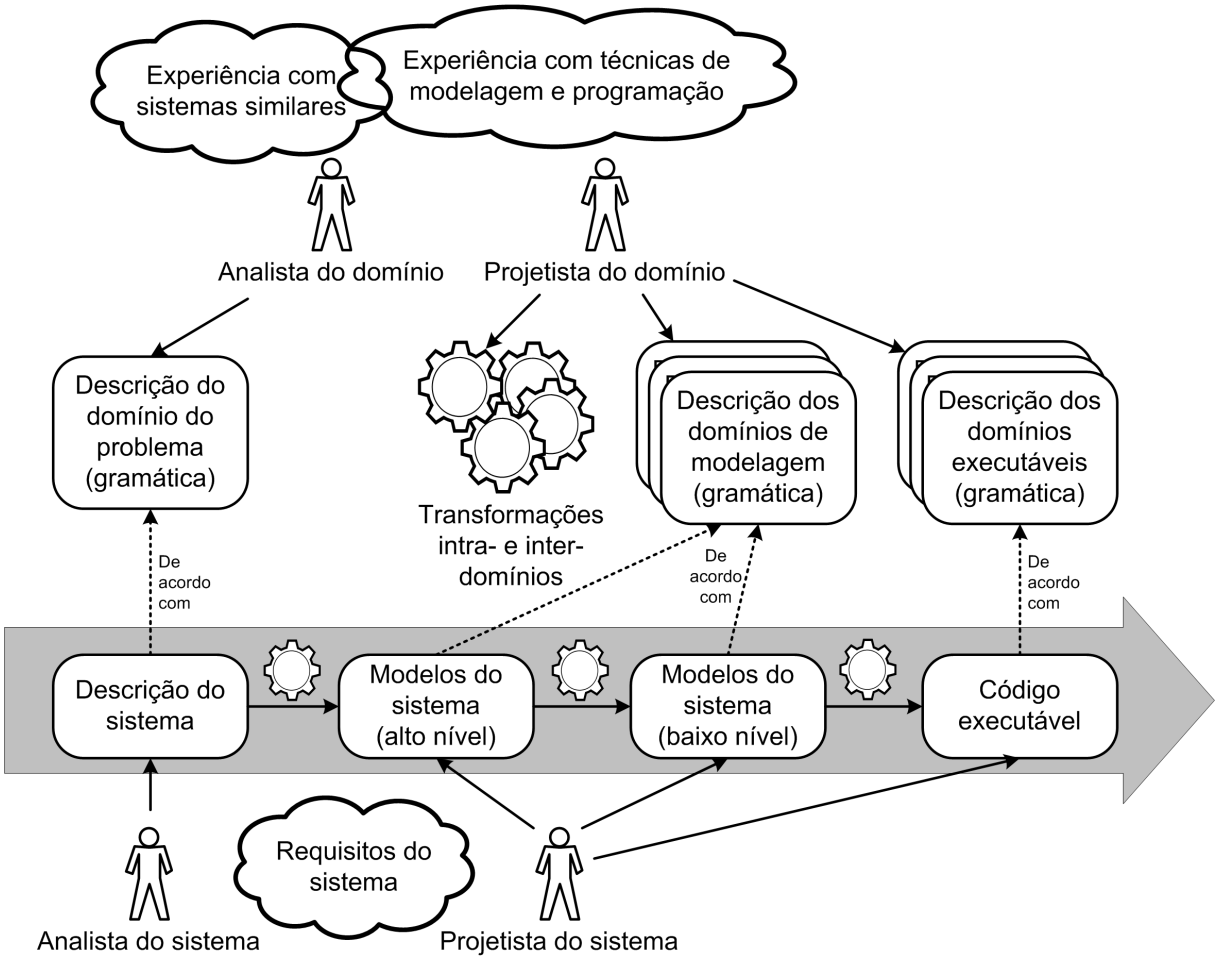


Figura 44: Processo de refinamentos sucessivos na abordagem Draco

O processo é dividido em duas fases distintas:

1. Na primeira fase, o analista do domínio, junto com o projetista do domínio, com base na experiência em sistemas similares e nas técnicas existentes para modelagem e programação, constróem um conjunto de domínios. O domínio mais abstrato, ou seja, aquele mais próximo da linguagem do especialista, é chamado de domínio do problema, ou domínio da aplicação. Os domínios intermediários, responsáveis pelos refinamentos até o nível executável, são chamados de domínios de modelagem. No nível mais baixo,

correspondente às linguagens de programação, estão os domínios executáveis. Nessa primeira fase, também são definidas as transformações de um domínio para outro; e

2. Na segunda fase, o analista de sistemas, com base nos requisitos de um sistema, cria uma descrição do sistema, de acordo com a linguagem do domínio do problema ao qual o sistema pertence. Essa descrição é automaticamente transformada para descrições nas linguagens de modelagem intermediárias, refinando os modelos, os quais o projetista pode modificar para inserir requisitos não funcionais. Os refinamentos se sucedem até produzir o sistema executável.

Esse processo é praticamente o mesmo para as abordagens de desenvolvimento orientado a modelos utilizadas atualmente, como por exemplo a abordagem definida nesta tese. A diferença está principalmente nas ferramentas para modelagem, criação e execução das transformações, já que atualmente essas são baseadas nos conceitos de metamodelagem, e não no uso de gramáticas. Este, inclusive, pode ser um dos motivos pelos quais a abordagem Draco não se estabeleceu de fato como uma abordagem para o desenvolvimento de software orientado a modelos, uma vez que a construção de gramáticas e transformadores baseados em regras gramaticais e compiladores mostrou-se muito complexa para ser aplicada na prática (WEIS; ULBRICH; GEIHS, 2003).

Além disso, as abordagens atuais possuem técnicas mais apuradas para captura e representação da variabilidade, além de mecanismos de implementação mais consolidados. A abordagem definida nesta tese define, por exemplo, padrões de projeto com o objetivo de facilitar o gerenciamento da variabilidade no contexto do MDD.

Outro diferencial desta tese com relação ao trabalho de Neighbors é uma preocupação maior com as atividades necessárias para o gerenciamento de múltiplos subdomínios automatizados. No Draco, a idéia de múltiplos domínios está mais intrinsecamente ligada a conceitos de linguagens de programação e de modelagem, e não com a divisão natural de um domínio de aplicações, como nesta abordagem. Assim, a identificação de áreas de interesse específicas para cada especialidade no domínio fica mais restrita à existência de linguagens específicas daquele domínio. Já nesta abordagem, um subdomínio pode ser identificado mesmo sem a existência de linguagens e/ou ferramentas dedicadas.

Após o trabalho de Neighbors, diversos esforços foram gastos em busca de uma melhor utilização dos conceitos do MDD. Recentemente foi desenvolvido o projeto *ModelWare*, como parte do IST (*Information Society Technologies*), uma das prioridades do planejamento estratégico envolvendo a pesquisa tecnológica realizada pela comunidade européia. Envolvendo

instituições de pesquisa e empresas de toda a Europa, o projeto *ModelWare* foi dividido em seis frentes de trabalho, descritas a seguir:

- **Tecnologias de modelagem**: o objetivo dessa frente é prover a base teórica e tecnológica de suporte para os desafios do MDD na indústria. Envolve a definição de mecanismos para descrever arquiteturas e plataformas, transformações de modelos, simulação de execução de modelos, assim como mecanismos para especificar e empacotar componentes MDD. Dentre os principais resultados alcançados por essa frente de trabalho, destacam-se os perfis para modelagem de arquiteturas e plataformas (MODELWARE, 2006a), e a definição do que seriam transformações reutilizáveis (MODELWARE, 2006b);

- **Processos e metodologias**: o objetivo dessa frente de trabalho é prover um conjunto de práticas para engenharia e gerenciamento de um processo de desenvolvimento orientado a modelos. Dentre os resultados dessa frente, destaca-se um *framework* de processo para o MDD (MODELWARE, 2006e) e um modelo de maturidade MDD (MODELWARE, 2006d);

- **Infraestrutura ferramental**: nessa frente de trabalho, foram desenvolvidas ferramentas e ambientes para uma abordagem de desenvolvimento orientado a modelos. A infraestrutura é baseada em código aberto, contemplando as questões de integração de ferramentas (MODELWARE, 2006g) e mecanismos de transformação (MODELWARE, 2006f);

- **Adoção bem sucedida**: uma das preocupações desse projeto é garantir que as tecnologias desenvolvidas sejam disseminadas e utilizadas na prática pela indústria. Neste sentido, foram realizadas, como parte dessa frente de trabalho, demonstrações e propostas para padronização (MODELWARE, 2006h);

- **MDD industrial**: essa frente de trabalho tem como objetivo validar os resultados obtidos pelas outras frentes de trabalho, na indústria; e

- **Gerenciamento**: diz respeito à manutenção da estratégia do projeto em conformidade com o contrato inicial e com os objetivos inicialmente previstos.

O projeto *ModelWare* é grande e abrangente, com resultados expressivos, tais como o MOFScript e ATL, descritos na Seção 2.2.1.

Esta tese se relaciona principalmente com a frente de processos e metodologias. O *framework* de processo e modelo de maturidade MDD possuem grande correlação com algumas atividades da abordagem definida nesta tese. Uma comparação mais detalhada com o modelo de maturidade encontra-se no Apêndice B.

Weis, Ulbrich e Geihs (2003) apresentam uma proposta de um mecanismo para modelagem, denominado *Kase*, e uma linguagem para transformação de modelos, denominada *Kafka*. Eles apresentam os requisitos a serem atendidos por transformações no desenvolvimento orientado a modelos. O primeiro deles é que as transformações devem ser automáticas, pois de outra forma os desenvolvedores considerariam a criação de modelos uma tarefa improdutiva. Outro requisito é que as transformações não podem ser universais e genéricas, mas sim específicas para cada caso, podendo ser adaptadas facilmente e reutilizadas. Finalmente, os autores ressaltam a necessidade de se utilizar linguagens visuais para a construção das transformações, uma vez que o uso de linguagens textuais, tais como linguagens de programação e linguagens de transformação baseadas em XML, possuem uma distância conceitual em relação aos modelos, dificultando a tarefa de construção de transformadores. As transformações representadas na linguagem *Kafka* são baseadas em regras de mapeamento, facilitando a chamada engenharia "ida e volta", ou seja, modelo para código e código para modelo.

A abordagem desta tese também busca automatizar completamente as transformações, de modo que não é necessário trabalho manual, o que poderia inibir o uso dos transformadores.

Outro aspecto similar é que tanto no trabalho de Weis, Ulbrich e Geihs (2003) como nesta tese, as transformações devem ser específicas para cada caso. A idéia aqui defendida é que os transformadores devem ser customizados de acordo com os requisitos e com o contexto do domínio sendo construído, da mesma forma que no mecanismo *Kase*.

Existem dois pontos, no entanto, nos quais a presente abordagem difere do *Kase*:

1. Esta abordagem não está restrita a linguagens visuais, por entender que há subdomínios onde o nível de abstração da especificação deve ser mais baixo, aproximando-se das características e poder expressivo de uma linguagem textual; e

2. Nesta tese, as transformações não se baseiam somente em regras de mapeamento, por dois motivos: (i) transformadores baseados em *templates* são mais simples de se construir e manter; e (ii) a necessidade da engenharia "ida e volta" é reduzida com o uso de uma implementação de referência e um processo de migração de código. Apesar de mais trabalhoso, esse caminho foi preferido por ser mais prático.

Czarnecki et al. (2005) discutem a combinação das idéias de linhas de produto (Seção 2.1.2) e desenvolvimento orientado a modelos, ressaltando que essas duas linhas são complementares. Neste sentido, os autores propõem a combinação da modelagem de características, ou *features*, com o uso de *templates* de características, responsáveis pelo mapeamento automático das características para modelos que as implementam, com base em transformações de modelos.

Um *template* de modelo baseado em *features* consiste em modelos de *features* e modelos anotados que implementam as *features*. Cada anotação enriquece um modelo com informações referentes às *features*, e podem ser na forma de condições de presença/ausência, iterações para expansão de *templates* ou expressões mais complexas para cálculo de elementos de modelagem. Os modelos anotados então representam como as configurações de um produto de uma linha influenciam na implementação da variabilidade correspondente. Uma variante consiste na instanciação das anotações, e desta forma, é possível gerenciá-la individualmente, de forma isolada.

A abordagem desta tese utiliza o mesmo tipo de construções para implementação das variabilidades, porém, ao invés de *templates* de modelos, aqui são utilizados principalmente *templates* de código, pois o objetivo é produzir implementações de forma mais direta, e não através de modelos intermediários. Além disso, nesta tese são definidas atividades para a construção dos templates e ligação não somente com modelos de *features*, mas também DSLs que representam variabilidades mais complexas em partes do domínio e/ou subdomínios.

Knodel et al. (2005) apresentam uma abordagem para a utilização de desenvolvimento orientado a modelos em linhas de produtos de software, chamada Pulse-MDD. O ponto central desta abordagem é o foco na arquitetura da linha de produtos como objetivo do processo de MDD, e não na plataforma de destino (como J2EE, por exemplo). A diferença é que com foco na arquitetura e nos produtos da linha, obtém-se maior grau de adequação ao domínio desejado, evita-se a ocorrência de problemas relacionados com a necessidade de se oferecer suporte a possibilidades mais genéricas que podem nunca ser necessárias à organização, facilitando o desenvolvimento.

O Pulse-MDD define atividades, realizadas como parte do projeto arquitetural, para a construção de geradores. Estas atividades buscam identificar padrões recorrentes no domínio, para serem utilizados como base para os geradores. Visando obter um conjunto economicamente otimizado de padrões, um processo iterativo identifica um conjunto de padrões e avalia a porcentagem de código gerado alcançado a cada iteração. Em um certo momento, torna-se muito complexo identificar e implementar um padrão na forma de geradores, e portanto o processo termina.

Esta abordagem possui muitas similaridades com a abordagem desta tese, que também é iterativa, mas especialmente com relação ao fato de que o desenvolvimento de transformações e ferramentas de modelagem é altamente ligado à arquitetura da linha de produto, e não em conceitos gerais das tecnologias de implementação. Os resultados são portanto mais próximos às necessidades organizacionais. O processo de criação dos geradores é também similar, com

base em exemplos do código a ser gerado, com a diferença de que no Pulse-MDD ele é guiado pela identificação de padrões e critérios econômicos, enquanto nesta tese o processo está centrado nos diferentes tipos de variabilidade.

A principal diferença desta tese com relação ao Pulse-MDD é que neste último a preocupação com o MDD começa somente na fase de projeto, enquanto nesta abordagem argumenta-se que ela deve começar antes, durante a análise, uma vez que os modelos de *features* possuem papel importante nesta definição, e normalmente precisam ser adaptados para refletir a existência dos artefatos MDD. Além disso, iniciando-se na análise, a preocupação com o MDD pode incluir a identificação e divisão de subdomínios automatizáveis. Segundo defende-se nesta tese, reconhecer esta divisão natural do domínio é um requisito importante para possibilitar o uso do MDD em cenários práticos.

Deelstra et al. (2003) discutem como uma abordagem para o desenvolvimento de linhas de produtos pode se beneficiar do desenvolvimento orientado a modelos. Os autores argumentam que o MDD pode levar a vários benefícios, e apresentam como ele pode ser utilizado para representação da variabilidade e derivação automática de produtos. O suporte à variabilidade com o MDD é através de modelos do domínio, a exemplo da abordagem desta tese. Porém, os autores passam diretamente da representação da variabilidade à derivação de produtos, sem entrar em detalhes sobre, por exemplo, a construção de uma arquitetura adequada para este cenário, como é o caso desta tese.

O mesmo acontece com outros trabalhos encontrados na literatura. Muitas abordagens que combinam o MDD com engenharia de domínio (ou linhas de produtos de software) possuem foco maior no estágio de derivação automática de produtos, dando pouca ou nenhuma ênfase ao projeto do domínio de acordo com os princípios do MDD. Exemplos destas abordagens incluem (WEISS et al., 2008; VÖLTER; GROHER, 2007; BOTTERWECK; O'BRIEN; THIEL, 2007; ARBOLEDA; CASALLAS; ROYER, 2007; TRASK et al., 2006; TOLVANEN; KELLY, 2005; CZARNECKI; HELSEN; EISENECKER, 2004b).

Uma das exceções é o método Kobra (ANASTASOPOULOS; ATKINSON; MUTHIG, 2002). Originalmente projetado para desenvolvimento de sistemas individuais, o Kobra se integra bem a um método voltado para linhas de produtos, como o Pulse-DSSA (ANASTASOPOULOS; ATKINSON; MUTHIG, 2002). O Kobra possui atividades específicas para o projeto arquitetural orientado a modelos. Os autores argumentam que dessa forma é possível obter independência de plataforma, pois o Kobra se baseia em modelos UML para especificação, realização e implementação de componentes de software no contexto de uma linha de produtos.

Assim como a abordagem desta tese, o Kobra utiliza cenários para viabilizar a criação

e avaliação de uma arquitetura para uma linha de produtos de software. Um diferencial do Kobra é que ele apresenta critérios para priorização de cenários, tais como o seu valor econômico, esforço necessário, entre outros. Nesta tese, a priorização é aberta, ficando a cargo dos *stakeholders* definirem aqueles mais importantes a serem utilizados como diretrizes arquiteturais.

No Kobra, são utilizados modelos UML, principalmente modelos de classes e de atividades, para especificação dos componentes, em diversos níveis de refinamento. Os autores sugerem a utilização de diagramas organizados de forma hierárquica, onde cada diagrama descreve um único componente. A hierarquia define o relacionamento entre eles, mostrando o domínio em diversos níveis, desde o contexto de sistema até os componentes individuais. Regras de relacionamento entre os diagramas definem de forma não ambígua como a relação entre os componentes deve ser implementada. Esta abordagem reflete o fato de que o Kobra é fortemente centrado no conceito de componentes de software, e como resultado a arquitetura e implementação estão descritos primariamente em termos dos seus componentes.

Diferentemente do Kobra, nesta tese o uso de componentes não é necessariamente obrigatório como mecanismo de encapsulamento de conhecimento e implementação da variabilidade. Unidades de implementação de mais baixo nível, como classes, podem existir tanto no nível arquitetural como de implementação. Desta forma, acredita-se que é possível oferecer suporte mais flexível e detalhado à variabilidade do domínio, para os casos onde componentes apresentam nível de granularidade muito alto.

Outra diferença entre esta tese e o método Kobra é que neste último o paradigma do MDD é somente parcialmente suportado, porque o método só lida com a atividade de modelagem, não cobrindo as atividades de desenvolvimento de transformações e geradores automatizados. Em contraste, o desenvolvimento de tais artefatos é explicitamente definido na abordagem desta tese, por meio de princípios, diretrizes e atividades específicas.

Blois (2006) desenvolveu uma abordagem de projeto arquitetural baseada em componentes no contexto de engenharia de domínio, cujo objetivo é atender aos principais requisitos para possibilitar um processo adequado para a reutilização de software. Segundo a autora, uma abordagem para engenharia de domínio baseada em componentes deve obedecer aos seguintes critérios: (i) Modelagem de variabilidades e opcionalidades em diferentes níveis de abstração; (ii) Manutenção da ligação e consistência entre os artefatos do domínio de diferentes níveis de abstração; (iii) Organização dos artefatos para a composição de elementos arquiteturais; (iv) Uso de princípios de DBC na modelagem de domínio; (v) Necessidade de processos de engenharia de domínio com apoio ao DBC; (vi) Geração de código fonte na linguagem de

programação empregada por uma tecnologia de DBC; e (vii) Modelagem e reutilização de artefatos e modelos através de ferramental apropriado. Segundo essa pesquisa, os principais métodos de ED e DBC não atendem a todos estes requisitos.

Assim como o Kobra, o trabalho de Blois (2006) é fortemente baseado no conceito de componentes de software, ou seja, visa produzir artefatos reutilizáveis que, segundo a definição de Sametinger (1997) utilizada, são *"autocontidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras, documentação apropriada e um grau de reutilização definido"*. Como resultado deste foco, a implementação obtida com a abordagem está também fortemente associada a uma tecnologia de componentes, como EJB (*Enterprise JavaBeans*), por exemplo. A autora propõe, além de um processo de ED com foco em DBC, uma notação similar à UML para representação da variabilidade no domínio, que vai além da mais comumente utilizada notação de *features*, podendo também ser utilizada em outros modelos, como modelos de classes, casos de uso e componentes. Também estabelece heurísticas de apoio à atividade de mapeamento entre as características do domínio e os demais artefatos de análise e projeto, além de critérios para agrupamento de componentes, visando a geração de elementos arquiteturais. O processo é apoiado por ferramentas que auxiliam sua execução e aplicam técnicas da MDA para geração automática de código.

A abordagem desta tese tem os mesmos objetivos de reutilização que o trabalho de Blois (2006), porém não se restringe a uma tecnologia de componentes. Aqui o foco é reconhecer uma maior diversidade de tecnologias de implementação e subdomínios, buscando oferecer suporte do MDD para produzir uma implementação que aproveite os diferentes potenciais de automação de cada subdomínio. Por estar direcionado a uma plataforma de implementação mais homogênea, o trabalho de Blois (2006) possui a vantagem de definir de forma mais sistemática suas atividades, principalmente com relação ao projeto da arquitetura de referência e a geração de código. Já nesta tese, as atividades são mais vagas, exigindo maior trabalho de adaptação por parte do engenheiro de software. Por exemplo, não há heurísticas para o mapeamento entre artefatos de análise e projeto e nem para auxiliar na identificação dos elementos arquiteturais. Além disso, são necessárias diversas atividades exclusivamente dedicadas ao projeto de DSLs, transformações e geradores, pois aqui a plataforma de implementação pode ser qualquer uma, e não necessariamente uma plataforma de componentes. Em contrapartida, a abordagem desta tese possui maior potencial de automação, com uma série de atividades dedicadas exclusivamente ao gerenciamento dessa diversidade dentro de um domínio.

Na realidade, ambas as abordagens são de certa forma complementares. Uma vez identificados os subdomínios, pode-se analisar onde o DBC deve ser utilizado, e então aplicar as técnicas propostas por Blois (2006) para mapear as suas características e projetar esta parte

da arquitetura do domínio utilizando uma tecnologia de componentes. A integração desses componentes com o restante do domínio pode então ser realizada conforme definido nesta tese.

Alferez et al. (2008) apresentam uma abordagem orientada a modelos para a engenharia de requisitos em um contexto de linha de produtos. Assim como na abordagem desta tese, os autores defendem a idéia de que as *features* devem ser complementadas com modelos adicionais (casos de uso e atividades) para melhor representar a variabilidade. Estes são utilizados para representar o comportamento variável associado com diferentes *features*. Em particular, busca-se refinar os modelos de caso de uso de forma que cada um esteja associado a somente uma *feature*, visando facilitar e automatizar as tarefas de engenharia de requisitos durante a configuração de produtos. Adicionalmente, a rastreabilidade entre as *features* e os demais modelos de requisitos é armazenada de acordo com um metamodelo dedicado.

Regras de composição entre os casos de uso são utilizadas para identificar os pontos de variação, de maneira similar ao conceito de casos de mudanças utilizado nesta tese. Uma regra de composição identifica o que muda, em termos de comportamento, de um caso de uso para outro. Como cada caso de uso está associado a uma *feature*, e as *features* são rastreáveis aos casos de uso e modelos derivados, estas regras permitem a geração automática dos modelos de requisitos de acordo com as *features* selecionadas para um determinado produto.

Nesta tese, a variabilidade dos cenários é representada por meio de casos de mudança, que constituem cenários isolados, e não informações sobre o que muda de um cenário para outro. Apesar de não possibilitar a automação, como no caso da abordagem de Alferez et al. (2008), o uso de cenários individuais facilita a sua utilização como diretrizes arquiteturais e a avaliação arquitetural. Foi por este motivo, para facilitar o projeto arquitetural, que a abordagem de casos de mudança foi adotada nesta tese.

Outra diferença é que a abordagem de Alferez et al. (2008) busca aplicar MDD para geração de modelos de requisitos para serem utilizados na fase de desenvolvimento de aplicações, sendo portanto mais completa no contexto de linhas de produtos. Nesta tese, o foco é somente na engenharia de domínio, sendo que a fase de desenvolvimento de aplicações não é abordada. Por este motivo, o uso do MDD se restringe somente à geração de implementação, não passando pelos passos anteriores de análise e projeto das aplicações.

O trabalho de Pérez-Martínez e Sierra-Alonso (2006) descreve uma abordagem para a derivação automática de uma arquitetura de software a partir de modelos de análise, utilizando transformações modelo-para-modelo semi-automatizadas. São definidas 32 regras de mapeamento que transformam casos de uso, classes, atributos e relacionamentos de modelos de análise, em modelos arquiteturais seguindo o estilo arquitetural conhecido como C2

(PÉREZ-MARTÍNEZ; SIERRA-ALONSO, 2006).

A exemplo dos geradores baseados em *templates* utilizados nesta tese, o mapeamento proposto por Pérez-Martínez e Sierra-Alonso (2006) utiliza regras imperativas, mais simples de implementar, mas mais complicadas quando se deseja promover rastreamento e realizar a transformação ida-e-volta.

Os modelos de destino são descritos utilizando um perfil UML2 desenvolvido exclusivamente para o C2, uma vez que a UML sozinha não é capaz de representar todos os elementos deste estilo, como por exemplo conectores, portas, mensagens, etc.

Diferentemente desta tese, porém, o trabalho de Pérez-Martínez e Sierra-Alonso (2006) assume que os modelos de análise são ricos o suficiente para serem automaticamente transformados em uma arquitetura. A abordagem desta tese defende a idéia de que é necessário trabalho manual e criativo considerável entre análise e projeto arquitetural, de forma a incluir as preocupações de todos os *stakeholders*. Assumindo-se um único e pré-definido mapeamento, corre-se o risco de se projetar uma arquitetura incapaz de atender a todos os requisitos de forma adequada. Além disso, um único estilo arquitetural (C2) é suportado.

Finalmente, a abordagem de Pérez-Martínez e Sierra-Alonso (2006) é focada no desenvolvimento de produtos únicos, não cobrindo aspectos relativos à reutilização, como suporte à variabilidade e derivação de produtos, como é o caso desta tese.

Völter e Groher (2007) descrevem um conjunto de técnicas para combinar MDD e linhas de produtos de software. Suas idéias sobre a combinação de técnicas estão de acordo com a filosofia da abordagem desta tese, tais como o suporte aos dois tipos principais de variabilidade: baseada em *features* (configuração) e DSLs (construção criativa). Adicionalmente, os autores utilizam técnicas orientadas a aspectos para facilitar a modelagem e implementação das variabilidades ortogonais, o que não é tratado nesta tese. A principal diferença desta tese em relação ao trabalho de Völter e Groher é que aqui é proposta uma abordagem sistemática, com atividades e diretrizes concretas para desenvolver a infraestrutura de reutilização orientada a modelos para o domínio. Além disto, a abordagem desta tese também identifica a necessidade do suporte simultâneo a múltiplos subdomínios, o que não é mencionado por Völter e Groher.

Robbes e Lanza (2008) descrevem um sistema de suporte à transformação de programas baseada em exemplos. Neste sistema, o programador realiza um exemplo de mudança manualmente, que é então submetido ao sistema e posteriormente generalizado para outros contextos, tornando-se uma transformação reutilizável. O sistema baseia-se no monitoramento do ambiente de programação para detectar automaticamente as modificações feitas pelo desenvolvedor em algum artefato de código. Estas são então registradas como operações de

mudanças, sendo posteriormente convertidas em entidades de primeira classe que descrevem como os elementos da AST (*Abstract Syntax Tree* ou Árvore de Sintaxe Abstrata) de um programa (como as classes, atributos, métodos e outras construções da linguagem) são modificados pelo desenvolvedor.

No passo de generalização, o sistema tenta identificar automaticamente o papel de cada mudança na transformação. Por exemplo, se uma mudança modifica um pedaço de código envolvendo-o com comandos *try-catch*, o trecho de código a ser envolvido é um parâmetro da transformação, enquanto os comandos *try-catch* são constantes a serem inseridas. O desenvolvedor da transformação pode então refinar esta generalização em um editor personalizado, renomeando parâmetros, especificando detalhes não detectados pelo sistema automático, entre outras tarefas.

O trabalho de Robbes e Lanza é focado principalmente na transformação de programas e *refactorings*. Porém, a abordagem poderia ser aplicada para linguagens de modelagem também. Como os próprios autores destacam, pode ser ainda mais simples nestes casos, já que os passos automáticos seriam realizados em estruturas mais simples do que uma AST de um programa. A abordagem desta tese é similar ao raciocínio de desenvolvimento de transformações através de exemplos seguido pelos autores, com a diferença de que a identificação das mudanças não é automatizada, sendo realizada completamente pelo desenvolvedor. Além disso, esta tese foca na engenharia de domínio, com o objetivo de produzir transformações projetadas especificamente para dar suporte à variabilidade no domínio.

Varró (2006) também propõe o desenvolvimento de transformações orientada a exemplos, porém com foco em transformações de modelos. Um processo iterativo e interativo utiliza derivação semi-automática das transformações, a partir de pares de modelos origem e destino. O desenvolvedor da transformação refina um conjunto de regras inicialmente derivadas até que as mesmas cheguem a um ponto satisfatório. Diferentemente do trabalho de Robbes e Lanza (2008) e da abordagem desta tese, o processo de Varró não depende do registro das modificações que levam da origem ao destino. Ao invés disso, o processo tenta estabelecer mapeamentos analisando exemplos de modelo origem e destino. A abordagem desta tese (assim como a de Robbes e Lanza (2008)) é mais voltada para os estágios iniciais do desenvolvimento, uma vez que tenta capturar o esforço manual para transformar modelos ou programas, à medida que os exemplos de destino são construídos, de forma incremental e gradual. Em contraste, o trabalho de Varró começa somente após exemplos de modelos de origem e destino já existirem, sendo portanto mais apropriado para estágios posteriores, quando o desenvolvedor já possui uma idéia concreta de como o destino da transformação deve ser.

Bierhoff, Liongosari e Swaminathan (2006) utilizam exemplos não somente para derivar as transformações, mas também as DSLs. Os autores propõem uma abordagem incremental: começando com uma aplicação, constrói-se uma DSL para ela, contendo os elementos e conceitos da aplicação, e os parâmetros necessários para configurá-la. Em seguida, novas aplicações são adicionadas incrementalmente, e suas diferenças introduzem novas variações na DSL, aumentando sua generalidade. Porém, os autores destacam que esta abordagem puramente *bottom-up* precisa ser orientada por decisões de projeto tomadas cuidadosamente de antemão, caso contrário corre-se o risco de se produzir DSLs extremamente complexas e difíceis de se manter. Este é exatamente o caminho tomado pela abordagem desta tese, que busca mesclar uma etapa *top-down* para preparar o domínio para automação, com uma etapa *bottom-up* que introduz detalhes e refinamentos necessários para o funcionamento prático das DSLs e das transformações.

Santos, Koskimies e Lopes (2008) propõem a extensão de *frameworks* com uma camada adicional baseada em programação orientada a aspectos que codifica uma DSL, buscando alavancar a reutilização de software através de técnicas generativas. Um modelo conceitual de um *framework* é manualmente extraído pelo desenvolvedor, que descreve seus elementos e conceitos em termos de um metamodelo específico. Este modelo é então utilizado em conjunto com um *framework* de linguagem para produzir ferramentas concretas para a nova linguagem. Utilizando estas ferramentas, o desenvolvedor de aplicações pode especificar uma aplicação e gerar código em termos dos elementos do *framework*. O trabalho de Muszynski (2005) também propõe a derivação de uma DSL a partir de uma implementação de referência, tornando a DSL extraída o ponto central para especificação de produtos e geração de código.

Estes trabalhos são similares à abordagem desta tese, pois utilizam uma DSL, geradores e um processo *bottom-up* para derivar os mapeamentos entre a linguagem e a implementação de referência (ou *framework*). Eles também possuem um elemento *top-down* implícito, que corresponde ao processo de desenvolvimento da implementação de referência ou do *framework*, o que envolve tarefas como gerenciamento de variabilidade na engenharia de domínio. A principal diferença é que na abordagem desta tese o elemento *top-down* é explícito e focado no problema da reutilização, indo mais além no processo, até o desenvolvimento de uma primeira versão da DSL de forma exclusivamente *top-down*. O resultado é que nesta tese as DSLs são mais próximas do domínio do problema, permitindo tarefas mais criativas, facilitando o desenvolvimento de aplicações e oferecendo mais flexibilidade ao desenvolvedor. Em contrapartida, o desenvolvimento de geradores é mais complexo devido a esta maior proximidade com o domínio do problema.

As extensões ao modelo de *features* descritas por Czarnecki, Helsen e Eisenecker (2004b),

com cardinalidades de *features* e atributos, entre outras (Seção 3.1.1), permitem a especificação de variabilidades mais complexas, de forma similar à variabilidade baseada em DSLs descrita na abordagem desta tese. Através destas extensões é possível representar praticamente o mesmo tipo de variabilidade que é possível representar utilizando-se, por exemplo, um metamodelo. A principal diferença é que com uma DSL tem-se um poder expressivo mais focado no problema, sendo portanto uma solução mais intuitiva, podendo inclusive ser utilizada e compreendida por especialistas. Mas nada impede que um modelo estendido de *features* seja convertido em um metamodelo, dando origem a uma DSL.

Mas além disso, a abordagem desta tese tem outras contribuições, como um conjunto de diretrizes e regras para ajudar no processo de identificação destas variabilidades, no desenvolvimento das sintaxes abstrata e concreta. Além disso, nesta tese o metamodelo é complementado com informações originadas em uma implementação concreta, o que facilita a identificação de variabilidades mais detalhadas e a construção de transformações e geradores de código mais precisos no suporte à variabilidade.

9.2 Trabalhos relacionados com o uso de métricas para MDD e reutilização

O estado da prática na avaliação de qualidade de modelos contém evidências de que a modelagem ainda é tida como uma atividade artesanal. Apesar de existirem alguns padrões e regras baseadas no bom senso (como minimizar o acoplamento, aumentar a coesão, manter uma hierarquia de classes pequena), os desenvolvedores ainda dependem muito da opinião de especialistas para determinar quando um modelo é bom ou não (FRANCE; RUMPE, 2007). Porém, existem diversos trabalhos que investigam a avaliação de modelos e o uso de métricas para aumentar a confiabilidade dos resultados da avaliação.

Mohagheghi e Aagedal (2007) apresentam os principais aspectos relacionados à avaliação de qualidade de um processo de MDD. Entre estes, destacam-se os aspectos de qualidade da linguagem de modelagem utilizada, tais como sua complexidade e adequação ao domínio, a qualidade das ferramentas utilizadas no processo, o conhecimento dos especialistas com relação ao uso das linguagens e ferramentas, a qualidade do processo utilizado e o uso de técnicas para identificar falhas e defeitos em projetos MDD. Nesta tese, foram utilizadas métricas para avaliação da linguagem de modelagem, como parte dos estudos de caso. Além disso, a qualidade do processo também foi uma preocupação importante, e a cobertura das atividades essenciais foi discutida na Seção 4.5, onde compara-se a abordagem desta tese com um modelo de maturidade em MDD.

Monperrus et al. (2008) argumentam que o custo para se coletar métricas em um projeto orientado a modelos é extremamente alto, devido à especificidade a um domínio, o que impede que sejam utilizadas métricas genéricas. Como consequência, é necessário desenvolver ferramentas específicas para medir software cada vez que um novo domínio é implementado. Para resolver este problema, os autores desenvolveram uma abordagem orientada a modelos para a geração de sistemas coletores de métricas específicas de domínio, com base em especificações formais das métricas a serem utilizadas. Especialistas do domínio especificam quais métricas são necessárias, seguindo um metamodelo definido pela abordagem, e são automaticamente geradas ferramentas capazes de coletar estas métricas.

Uma abordagem parecida é apresentada por Guerra, Lara e Díaz (2008). Neste trabalho, os autores descrevem uma linguagem específica de domínio para a definição de métricas. Esta linguagem pode servir de entrada para a coleta automática destas métricas. Porém, diferentemente do trabalho de Monperrus et al. (2008), aqui os autores acrescentam a preocupação com reprojeto e refatoração de modelos, visando a sua melhoria. A linguagem para definição de métricas também inclui ações que representam modificações nos modelos, de forma a implementar o reprojeto.

Nos estudos de caso desta tese, o objetivo foi avaliar a abordagem e suas vantagens com relação ao desenvolvimento para reutilização tradicional, e portanto não foi necessária a definição de nenhuma métrica específica para um domínio.

Genero, Poels e Piattini (2008) apresentam doze métricas para avaliação estrutural de modelos entidade-relacionamento, com o objetivo de determinar sua manutenibilidade através de sua compreensibilidade. A premissa de seu trabalho é a de que quanto mais compreensível for um diagrama, maior será a sua facilidade de manutenção. Foi realizado um estudo empírico, que demonstrou que a compreensibilidade de um diagrama E-R é afetada por sua complexidade estrutural, ditada pela quantidade de atributos e relacionamentos, em particular relacionamentos 1:1 e 1:N. Quanto mais atributos e relacionamentos (1:1 e 1:N) um diagrama possuir, menos compreensível ele será. É interessante notar que não foi identificada evidência que relaciona o tamanho de um diagrama em termos de número de entidades com a compreensibilidade, a não ser através de sua relação óbvia com o número de relacionamentos. Igualmente, não foi evidenciada relação com o número de relacionamentos N:M. Assim, entre as doze métricas originalmente propostas, apenas estas três foram comprovadamente verificadas como sendo relevantes para a manutenibilidade. Em outro trabalho (GENERO et al., 2007), alguns destes mesmos autores demonstram que as mesmas propriedades estruturais de diagramas E-R também possuem influência na manutenibilidade de diagramas de classes UML, particularmente os relacionamentos de associação e generalização. Estas métricas foram adaptadas e utilizadas

nesta tese, durante os estudos de caso, conforme descrito no Capítulo 8.

Pilgrim (2008) apresenta algumas métricas para determinar o nível de abstração de um modelo. O objetivo é avaliar a eficiência das transformações entre modelos, argumentando que transformações devem inicialmente ser focadas em transformações na semântica, e somente posteriormente devem ser incluídos detalhes da plataforma. As métricas se baseiam no número de atributos, a exemplo do trabalho de Genero, Poels e Piattini (2008), mas também no tamanho do diagrama. Nesta tese, não foram utilizadas estas métricas, pois a eficiência das transformações não foi o foco dos estudos de caso.

Chidamber e Kemerer (1994) apresentam algumas métricas clássicas da orientação a objetos baseadas em conceitos como acoplamento, coesão, complexidade de objeto, escopo de propriedades, conjunto de respostas e combinação de classes. Os autores definem as seguintes métricas: métodos ponderados por classe (WMC - *Weighted Methods per Class*); profundidade da árvore de herança (DIT - *Depth of Inheritance Tree*); número de filhos (*NOC - Number Of Children*); acoplamento entre classes (CBO - *Coupling Between Object classes*); resposta para uma classe (RFC - *Response For a Class*); e falta de coesão em métodos (LCOM - *Lack of Cohesion in Methods*). Estas métricas focam em diferentes aspectos de um projeto, tais como complexidade, dificuldade de desenvolvimento e manutenção. Os autores também destacam o papel do acoplamento na reutilização. Quanto menor o acoplamento, mais fácil será a reutilização de uma determinada classe. Outras métricas clássicas para projetos orientados a objetos incluem a *estabilidade* (MARTIN, 1994), *manutenibilidade* (VANDOREN, 1997) e *complexidade* (MCCABE, 1976).

Algumas destas métricas foram utilizadas nos estudos de caso, nas partes do domínio onde não houve aplicação do MDD e para comparação com o software construído manualmente, conforme descrito no Capítulo 8.

Muskens, Chaudron e Lange (2004) propõem a coleta de métricas clássicas da orientação a objetos diretamente em modelos UML, ou seja, antes que exista código-fonte. Os resultados mostraram que é possível detectar os mesmos problemas utilizando esta abordagem, porém no início do desenvolvimento. Portanto, ações corretivas são menos dispendiosas, uma vez que não exigem a modificação extensa do código. Os autores também propõem métricas adicionais baseadas em modelos arquiteturais descritos segundo a visão 4 + 1 (KRUCHTEN, 1995). O argumento é que as métricas clássicas, mesmo sendo coletadas a partir dos modelos UML, não consideram as interações entre as diferentes visões, como por exemplo a referência a classes e métodos a partir de um diagrama de sequência. Porém, a validade destas métricas não é apresentada neste trabalho. Nesta tese, não foi possível aplicar estas métricas, uma vez que elas

estão fixadas na UML e na visão 4 + 1, não sendo adequadas para DSLs.

Lange e Chaudron (2004) apresentam algumas regras e métricas para avaliação de modelos UML, considerando a sua completude e consistência. Exemplos destas regras incluem: *objetos sem nome*, *classes sem métodos*, *interfaces sem métodos*, *classes abstratas em diagramas de sequência*, *classes não chamadas em diagramas de sequência*, *mensagens entre classes não relacionadas*, entre outras que ajudam na avaliação e garantia da qualidade de modelos UML. Para esta tese estas métricas não são úteis, uma vez que não podem ser usadas para comparação dos níveis de reutilização com e sem a aplicação de MDD conforme proposto na abordagem.

Em outro artigo, Lange e Chaudron (2005) apresentam um modelo de qualidade para desenvolvimento de software baseado em UML. Além dos atributos de qualidade a serem avaliados, como complexidade, rastreabilidade, modularidade, comunicabilidade e estética, entre outros, os autores apresentam uma lista sobre quais métricas podem ser utilizadas para avaliar estes atributos. Por exemplo, para avaliar a complexidade, os autores citam as métricas de *profundidade da árvore de herança* (DIT), *coesão*, *número de casos de uso por classe* e *número de classes por caso de uso*. Nesta tese não foi feita avaliação de qualidade do software baseado em UML, porém algumas das métricas foram utilizadas nos estudos de caso, com o objetivo de avaliar os benefícios da abordagem proposta.

A iniciativa Modelware também investigou métricas para o desenvolvimento orientado a modelos (MODELWARE, 2006c). Foram definidas métricas para diversos aspectos de engenharia, incluindo métricas de qualidade do produto, incluindo qualidade dos modelos e demais artefatos, métricas de processo, incluindo transformações e geradores de código, e métricas de projeto, úteis para estimativas e outras tarefas de gerenciamento. É interessante notar que as métricas sugeridas para a qualidade da arquitetura são as mesmas da orientação a objetos descritas por Chidamber e Kemerer (1994): *WMC*, *RFC*, *LCOM*, *CBO*, entre outras. A exemplo do trabalho de Muskens, Chaudron e Lange (2004), estas levam à identificação dos mesmos problemas do que quando coletadas diretamente do código-fonte, com a diferença de que isto ocorre durante o projeto.

Mascena, Almeida e Meira (2005) e Mascena (2007) apresentam uma análise sobre métricas relacionadas à reutilização de software, incluindo três categorias: métricas orientadas a fatores econômicos, estruturais e de repositório. Entre estas, as métricas estruturais são a base para analisar de forma mais aprofundada *o que* está sendo reutilizado. São elas: porcentagem de reutilização (RP - *Reuse Percent*) (POULIN; CARUSO, 1993); nível de reutilização (RL - *Reuse Level*) (FRAKES; TERRY, 1994); frequência de reutilização (RF - *Reuse Frequency*) (FRAKES; TERRY, 1994); tamanho e frequência de reutilização (RSF - *Reuse Size and Frequency*)

(DEVANBU et al., 1996); razão de reutilização (RR - *Reuse Ratio*) (DEVANBU et al., 1996); e densidade de reutilização (RD - *Reuse Density*) (CURRY et al., 1999). Estas são métricas simples, porém não podem ser consideradas como indicadores isolados, uma vez que possuem problemas por não considerar a natureza dos artefatos reutilizáveis e nem a maneira com que estes são reutilizados, penalizando, por exemplo, grandes sistemas e sistemas pouco modularizados (monolíticos). Por este motivo, existe outra vertente que defende a idéia de que é melhor tentar medir a reutilização através da avaliação de atributos de qualidade que medem o quão reutilizável é um determinado artefato de software. Neste sentido, são sugeridas métricas indiretas, como manutenibilidade e complexidade (POULIN, 1994). Estas já foram utilizadas com sucesso em outros estudos relacionados à reutilização de software, como por exemplo o trabalho de Almeida et al. (2007a). Nesta tese, foram utilizadas algumas das métricas de reutilização, além das métricas indiretas, na avaliação dos estudos de caso.

9.3 Considerações finais

A literatura nas áreas de reutilização de software e desenvolvimento orientado a modelos é relativamente rica em trabalhos que exploram os aspectos processuais destes dois paradigmas. Porém, a investigação conjunta das duas linhas de pesquisa ainda é recente, e normalmente os trabalhos focam em partes isoladas do problema. Apesar de esta tendência ser natural e, academicamente, mais seguro, ainda existem espaços para contribuições que resultem da investigação conjunta das duas linhas.

É interessante notar que as duas áreas, aparentemente distintas, tiveram um único pesquisador que investigou de forma pioneira diversos conceitos que posteriormente formaram a base de cada área separadamente. James Neighbors, em sua abordagem Draco (NEIGHBORS, 1980), combinou os conceitos de domínio (tendo cunhado o termo “Análise de domínio”) e transformações de software de forma inovadora. Esta união entre reutilização e MDD permaneceu relativamente ignorada por vários anos, sendo atualmente resgatada principalmente após o surgimento da MDA (LUCRÉDIO et al., 2006).

Neste capítulo foram apresentados trabalhos acadêmicos na área de desenvolvimento orientado a modelos e reutilização de software, e que possuem alguma interseção com a abordagem desta tese. Também foram discutidos trabalhos relacionados com a avaliação realizada, e que serviram de base para a definição das métricas utilizadas nesta tese.

10 Conclusões

Reutilização de software é um objetivo constantemente procurado por pesquisadores e profissionais envolvidos com desenvolvimento de software. A percepção comum é a de que esta é uma idéia bastante simples de se colocar em prática, que apresenta benefícios consideráveis e praticamente não requer investimento em tecnologias ou profissionais altamente qualificados.

Décadas de pesquisa evidenciaram que, apesar de ser uma idéia simples, a sua aplicação na prática é algo extremamente complexo, principalmente de forma sistemática e gerenciável. Muitos fatores fogem da área técnica, e portanto é mais difícil para profissionais desta área perceberem os erros e dificuldades na sua aplicação. Como resultado, a reutilização vem ocorrendo, porém de forma isolada e heróica, quando deveria ser uma prática mais disseminada na comunidade.

Mesmo no âmbito técnico, a reutilização ainda esbarra em problemas do desenvolvimento de software baseado em código-fonte. É cada vez mais difícil construir software atualmente, dada a sua complexidade e ubiquidade, ambas causadas pelo progresso tecnológico. Enquanto atualmente é possível resolver problemas através de *frameworks*, padrões e outras técnicas, eventualmente chegará o dia em que nossa capacidade de construir software não será capaz de atender à demanda. Assim como em outras indústrias, a automação pode aumentar esta nossa capacidade, suprindo nossas deficiências e limitações relacionadas ao desenvolvimento de software.

10.1 Principais contribuições

Nesta dissertação busca-se resolver parte do problema da reutilização através do desenvolvimento orientado a modelos. Após estudos e pesquisas nas áreas relacionadas, foi definida uma abordagem orientada a modelos para reutilização de software, visando guiar o engenheiro de software desde as atividades iniciais de análise até a implementação de artefatos reutilizáveis de um domínio, utilizando o desenvolvimento orientado a modelos para elevar e melhorar a reutilização. Neste sentido, as seguintes contribuições foram feitas nesta área:

- Uma abordagem sistemática e bem definida, contendo atividades, entradas e saídas que detalham as tarefas necessárias para que a engenharia de domínio possa incorporar as técnicas do desenvolvimento orientado a modelos. Em particular, foi definido um método concreto para identificação de múltiplos subdomínios, incluindo diretrizes de apoio e um processo investigativo, interativo e iterativo, adequado à natureza incerta e evolutiva desta área;

- Identificação de um conjunto de diretrizes e padrões arquiteturais focados especificamente nos problemas da reutilização e desenvolvimento orientado a modelos, visando possibilitar o gerenciamento de múltiplos subdomínios durante o projeto arquitetural;

- Realização de estudos empíricos. A literatura apresenta pouca evidência quantitativa sobre os benefícios do MDD às organizações de software (MOHAGHEGHI; DEHLEN, 2008). Assim, os resultados deste estudo são relevantes não somente para avaliar a abordagem definida nesta tese, mas também para a comunidade científica e industrial interessada em aplicar o MDD; e

- Elaboração de um material de treinamento em MDD, utilizado nos estudos de caso, mas que pode ser reaproveitado em cursos de extensão, minicursos e tutoriais em eventos relacionados.

10.2 Lições aprendidas

As contribuições citadas são de caráter científico e acadêmico, apresentando melhorias e aspectos ainda pouco explorados na literatura. Porém, com o desenvolvimento deste trabalho, diversas lições importantes foram aprendidas.

Durante esta pesquisa, foram estudadas diferentes tecnologias relacionadas ao MDD. Neste período, pode-se perceber a evolução do estado-da-arte e da prática nesta área. Há poucos anos atrás, a falta de ferramentas e suporte para este tipo de desenvolvimento era uma forte preocupação. A proposta inicial deste trabalho, apresentada em 2005, previa o desenvolvimento de parte das ferramentas necessárias para a aplicação do MDD. Na época em que esta dissertação foi escrita, em 2009, não só existem ferramentas disponíveis, mas também existem diversas opções, todas elas sendo efetivamente utilizadas na prática pela indústria. Nesta tese, foram utilizadas três alternativas diferentes, e ainda existem muitas outras igualmente capacitadas. Isto demonstra a tendência de que o MDD está – e será cada vez mais – presente na realidade do desenvolvimento de software.

A realização dos estudos empíricos também proporcionou o aprendizado de importantes lições. A primeira delas foi a importância da experiência em ambiente industrial. Em ambas as experiências, percebeu-se que apesar do crescimento em importância, o MDD ainda é uma tecnologia muito distante da realidade da grande maioria das empresas. Este fato pode ser percebido no terceiro estudo, onde a empresa envolvida sequer conhecia a maioria dos conceitos envolvidos.

E este não parece ser um retrato somente do cenário brasileiro. Por exemplo, o segundo estudo empírico, realizado durante o estágio de doutorado na Microsoft Research, foi uma das primeiras iniciativas em MDD deste importante instituto de pesquisa. Durante as apresentações e seminários no final do estágio, realizadas pelo autor desta tese, muitos dos participantes ali presentes estavam vislumbrando os conceitos do MDD pela primeira vez, inclusive importantes pesquisadores e funcionários desta grande empresa da área de TI. O fato de que a própria Microsoft possui uma solução voltada para o desenvolvimento de DSLs e geradores de código apenas reforça esta observação. Claro que não se pode generalizar estas observações com base em fatos isolados, mas a experiência destes anos de doutorado, após participações em diversos eventos científicos e contatos com empresas, ainda não produziram evidência do contrário.

Outra importante lição aprendida diz respeito à grande variedade de trabalhos similares existentes na literatura. É muito comum encontrar trabalhos bastante parecidos com os temas aqui investigados. Porém, há também grande diversidade no foco, com cada trabalho sendo bastante direcionado a uma determinada linha de pesquisa característica do grupo de pesquisa que o realiza. Não existem muitos trabalhos derivados destas iniciativas isoladas e que buscam englobar e unificar os diversos aspectos do problema, caminho este que foi seguido nesta tese. As contribuições alcançadas comprovam que há espaço também para este tipo de pesquisa.

Neste quesito, cabe também uma observação: a despeito da grande quantidade e variedade de trabalhos na área, é raro encontrar estudos empíricos buscando sua validação, a exemplo do que se tentou realizar nesta pesquisa. É difícil encontrar métricas e valores para serem utilizados como base para este tipo de estudo, sendo frequentemente necessárias adaptações ou interpretações indiretas para se extrair conclusões relevantes.

10.3 Limitações e Trabalhos futuros

Foram também identificadas, como subproduto desta pesquisa, oportunidades para trabalhos futuros, seja para complementar esta pesquisa explorando limitações e/ou aspectos que não puderam ser investigados nesta tese, ou para iniciar novas investigações em áreas

identificadas como sendo deficitárias de contribuições. Assim, os seguintes pontos podem ser investigados futuramente:

- **Desenvolvimento com reutilização:** a presente abordagem não prevê atividades para o desenvolvimento com reutilização. Apesar de os resultados da engenharia de domínio poderem beneficiar o desenvolvimento de software, há uma série de desafios envolvidos na reutilização dos produtos da ED. Assim, a sequência mais direta deste trabalho é a investigação do processo de desenvolvimento com reutilização, ou seja, como produzir aplicações que reutilizam os artefatos desenvolvidos com a abordagem desta tese de forma a maximizar a reutilização e fazer uso efetivo das linguagens, transformações e geradores de código produzidos. Neste cenário, um processo semelhante, envolvendo análise, projeto e implementação, deve ser realizado. Porém, este trabalho deve definir exatamente os passos e atividades necessários para que os esforços realizados durante a engenharia do domínio possam ser reutilizados, como por exemplo: ao realizar a análise dos requisitos, deve-se considerar que já existem modelos de *features*, casos de uso e de mudança para o domínio. Como identificar quais modelos do domínio se adequam aos requisitos da aplicação? Qual o procedimento a ser realizado caso exista um requisito que não foi incluído na análise do domínio? Vale a pena tentar renegociar com o cliente os requisitos da aplicação de forma a promover maior reutilização? Além disso, tanto o projeto como a implementação das aplicações devem considerar possíveis adaptações e/ou inserções de novos artefatos. Neste contexto, o controle de configuração deve ser uma atividade indispensável;

- **Replicação dos estudos empíricos:** outra limitação desta tese é a avaliação. Sabe-se que é difícil realizar experimentos em engenharia de software, visto que há uma série de fatores a serem controlados. Por exemplo, as métricas utilizadas não são uma medida precisa da reutilização e reusabilidade dos artefatos produzidos. Além disso, foi realizada somente uma avaliação dos produtos (artefatos) da abordagem, não havendo preocupação com a avaliação do processo. Todos estes fatores contribuem para que a validade dos resultados seja questionada. Assim, visando melhorar a validade das conclusões, pode-se realizar novos experimentos envolvendo a definição mais precisa dos seus elementos e operacionalizando-os, por exemplo, com equipes maiores. Estes trabalhos poderiam ser realizados, inicialmente, como iniciação científica, posteriormente evoluindo para um mestrado, visando incluir melhorias e correções na abordagem, com base nos resultados;

- **Reutilização de outros tipos de artefatos:** nesta tese não foi feita distinção com respeito ao tipo de artefato sendo gerado. Não há atividades ou diretrizes específicas para o

desenvolvimento de transformações que geram outras coisas além do código-fonte, como outros modelos, arquivos de configuração, documentação, entre outros. Este é um ponto interessante, pois neste tipo de artefato não há a possibilidade de utilização, por exemplo, de padrões de projeto e outras técnicas que se baseiam em conceitos da orientação a objetos, em particular, a herança. Este poderia ser o tema de um trabalho de mestrado;

- **Especialização da abordagem para domínios específicos:** outra limitação desta tese é que a abordagem é um pouco vaga em algumas atividades, como aquelas para a definição das DSLs, por exemplo. Idealmente, deveria-se buscar um processo mais sistemático, de forma a reduzir as possibilidades de erros e interpretações erradas. Porém, em alguns casos não foi possível definir mais detalhes pela própria natureza criativa do processo de criação de software. Trabalhos futuros, principalmente de mestrado, poderiam explorar a aplicação da abordagem em diferentes domínios, visando detalhar mais algumas atividades conforme as características destes domínios. Durante esta aplicação, poderiam ser identificadas heurísticas que facilitem principalmente as atividades de projeto arquitetural voltado para o MDD, construção de DSLs, transformações e geradores de código especializados para um determinado domínio;

- **Assistência contextualizada automatizada:** esta é outra possibilidade de trabalhos futuros. A abordagem não oferece nenhum tipo de ajuda no sentido de acompanhamento de processo, com base no contexto da tarefa atual ou de outras informações relevantes, como por exemplo a necessidade de aplicação de um padrão ou estilo arquitetural. Neste possível trabalho futuro, a partir de sugestões sobre quais atividades a serem realizadas em um contexto particular, como por exemplo a exibição automática de ajuda específica de contexto, desenvolvedores de produtos podem ser ativamente guiados durante tarefas de solução de problemas. No contexto do MDD, ela pode ajudar com as tarefas complexas envolvidas com o desenvolvimento das linguagens e ferramentas específicas de domínio, transformações e adaptação manual de código gerado. Tecnologias como o quadro de tarefas do GMF (*Generic Modeling Framework*), as receitas do *openArchitectureWare* (Seção 2.2.2) e as Microsoft Blueprints[1] são exemplos dessa assistência sendo aplicada ao MDD. Trabalhos futuros podem explorar as atividades necessárias para o desenvolvimento deste tipo de assistência no contexto do MDD e da engenharia de domínio. Este poderia ser o tema de um trabalho de mestrado e/ou doutorado, dependendo da sua abrangência;

- **Colaboração no MDD:** o desenvolvimento orientado a modelos possui diferentes tarefas

[1] `http://msdn.microsoft.com/en-us/architecture/blueprints.aspx`

que envolvem a participação de múltiplos membros de uma equipe, tais como a análise de *features* e a definição de múltiplas DSLs para diferentes domínios. Também pode-se citar a existência de equipes distintas para trabalhar com a implementação de referência, com as tecnologias de modelagem e os geradores de código. Porém, a determinação exata de onde a colaboração no MDD ocorre de maneira mais intensa, visando esclarecer os pontos de flexibilização na comunicação e atividades da equipe que podem ser alcançados, ainda se mostram incipientes, como indicado por Steffen et al. (2007). Assim, trabalhos futuros podem investigar a natureza desta colaboração, incluindo possivelmente a consideração sobre desenvolvimento distribuído, ou até mesmo os aspectos ferramentais envolvidos na colaboração. Este poderia ser o tema de um trabalho de mestrado e/ou doutorado, dependendo da sua abrangência; e

- **Migração automática de código:** o uso da implementação de referência como ponto de partida para o desenvolvimento de geradores traz uma série de benefícios. Porém, o processo de migração de código, necessário neste mapeamento para os geradores, apresenta dois problemas (MUSZYNSKI, 2005): (i) trata-se de um processo que consome cerca de 20-25% do tempo de desenvolvimento da implementação de referência; e (ii) causa duplicação de código, pois apesar de após a primeira migração ser possível descartar completamente a implementação de referência, na prática isto não ocorre devido às dificuldades de se trabalhar no nível do gerador. Como resultado, são mantidas tanto a implementação de referência como o gerador, e continua-se trabalhando com os dois artefatos. Apesar de não-trivial, a automação, mesmo que parcial, seria a solução para estes problemas. Ainda não existe uma solução automática para migração de código nas principais ferramentas do mercado, e este é um interessante tema para trabalhos futuros, podendo ser realizados como um pós-doutoramento na área.

10.4 Considerações finais

Nesta dissertação apresentou-se a tese de que o MDD pode aumentar e/ou melhorar a reutilização de software em projetos de engenharia de domínio, descrevendo uma abordagem para esta finalidade e sua avaliação através de estudos empíricos. A abordagem combina elementos de processo, como atividades, tarefas, entradas e saídas, com diretrizes e técnicas auxiliares que guiam o desenvolvedor durante a engenharia de domínio orientada a modelos. A principal contribuição está em reconhecer a existência de múltiplos subdomínios, cada um com seu potencial de automação, e em oferecer um método concreto para que o engenheiro de software possa lidar com eles.

Esta abordagem é um passo inicial em direção a um *framework* de processo completo para a utilização efetiva do MDD como facilitador da reutilização. Trata-se de um ponto central, ao redor do qual é necessário definir atividades de suporte e de gerência, além de outras práticas de engenharia não cobertas pela abordagem.

Esta dissertação apresenta os resultados de uma pesquisa aplicada, que busca unificar diferentes técnicas relacionadas à reutilização de software e MDD de uma forma original, oferecendo um suporte combinado que é melhor do que a aplicação das técnicas de forma isolada. Os resultados foram interessantes e esclarecedores, principalmente em termos da utilização prática em ambiente industrial. Comparando-se com o estado-da-arte, nota-se que as contribuições deste tipo de pesquisa são relevantes, tendo em vista a escassez de trabalhos que visam somente solidificar e formalizar contribuições que ainda apresentam falhas e falta de detalhes sobre as diversas formas de desenvolvimento produtivo de software.

Referências

ALFEREZ, M. et al. A model-driven approach for software product lines requirements engineering. In: *SEKE*. [S.l.]: Knowledge Systems Institute Graduate School, 2008. p. 779–784. ISBN 1-891706-22-5.

ALLILAIRE, F. et al. Global model management in Eclipse GMT/AM3. In: *Eclipse Technology eXchange Workshop (eTX) at the ECOOP 2006 conference*. Nantes, France: [s.n.], 2006.

ALMEIDA, E. S. C.R.U.I.S.E - Component Reuse In Software Engineering. In: ____. [S.l.]: C.E.S.A.R. e-books, 2007. cap. 4 - Software Reuse Processes: The State-of-The-Art, p. 81–100.

ALMEIDA, E. S. et al. An experimental study in domain engineering. In: *33rd IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Component-Based Software Engineering Track*. [S.l.: s.n.], 2007. p. 93–100.

ALMEIDA, E. S. et al. A systematic approach to design domain-specific software architectures. *Journal of Software - Academy Publisher*, v. 02, n. 2, p. 38–51, 2007.

ALMEIDA, E. S. et al. RiSE project: Towards a robust framework for software reuse. In: *IEEE International Conference on Information Reuse and Integration (IRI)*. Las Vegas, USA: IEEE/CMS, 2004. p. 48–53.

ALMEIDA, E. S. et al. A survey on software reuse processes. In: *IEEE International Conference on Information Reuse and Integration (IRI 2005)*. Las Vegas, USA: IEEE/CS Press, 2005. p. 66–71.

ALMEIDA, E. S. et al. The domain analysis concept revisited: A practical approach. In: *9th International Conference on Software Reuse (ICSR)*. Torino, Italy: Lecture Notes in Computer Science, Springer-Verlag, 2006. v. 4039, p. 43–57.

ALMEIDA, E. S. et al. Domain implementation in software product lines using OSGi. In: *7th International Conference on Composition-Based Software Systems (ICCBSS)*. Madrid, Spain: [s.n.], 2008. p. 72–81.

AMBLER, S. W. Agile model driven development is good enough. *IEEE Software*, v. 20, n. 5, p. 71–73, 2003.

ANASTASOPOULOS, M.; ATKINSON, C.; MUTHIG, D. A concrete method for developing and applying product line architectures. In: AKSIT, M.; MEZINI, M.; UNLAND, R. (Ed.). *NetObjectDays*. [S.l.]: Springer, 2002. (Lecture Notes in Computer Science, v. 2591), p. 294–312. ISBN 3-540-00737-7.

ANASTASOPOULOS, M.; GACEK, C. Implementing product line variabilities. In: *Symposium on Software Reusability (SSR)*. Toronto, Canada: [s.n.], 2001. p. 109–117.

ANTKIEWICZ, M.; CZARNECKI, K. Framework-specific modeling languages with round-trip engineering. In: NIERSTRASZ, O. et al. (Ed.). *Model Driven Engineering Languages and Systems (MoDELS 2006)*. [S.l.]: Springer Berlin / Heidelberg, 2006. (Lecture Notes in Computer Science, v. 4199/2006), p. 692–706.

ARANGO, G. Domain analysis - from art form to engineering discipline. In: *International Workshop on Software Specifications & Design*. Pittsburgh, Pennsylvania, United States: [s.n.], 1999. p. 152–159.

ARBOLEDA, H.; CASALLAS, R.; ROYER, J.-C. Dealing with constraints during a feature configuration process in a model-driven software product line. In: SPRINKLE, J. et al. (Ed.). *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM07) - Montréal, Canada*. [S.l.]: University of Jyväskylä, Finland 2007, ISBN 978-951-39-2915-2., 2007. (Computer Science and Information System Reports, Technical Reports, TR-38), p. 178–183.

BAHNOT, V. et al. Using domain-specific modeling to develop software defined radio components and applications. In: *The 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego USA*. [S.l.: s.n.], 2005. p. 33–42.

BARTHOLDT, J.; OBERHAUSER, R.; RYTINA, A. An approach to addressing entity model variability within software product lines. In: *ICSEA*. [S.l.]: IEEE Computer Society, 2008. p. 465–471. Disponível em: <http://dx.doi.org/10.1109/ICSEA.2008.30>. Acesso em: 14 jun 2009.

BASILI, V.; ABD-EL-HAFIZ, S. K. A method for documenting code components. *Journal of Systems and Software (JSS)*, v. 34, n. 2, p. 89–104, 1996.

BASILI, V. R.; BRIAND, L.; MELO, W. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, v. 39, n. 10, p. 104–116, 1996.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. In: *Encyclopedia of Software Engineering, Vol. II*. [S.l.: s.n.], 1994. v. 2, p. 528–532.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice, 2nd Edition*. [S.l.]: Addison-Wesley, 2003.

BAUER, D. A reusable parts center. *IBM Systems Journal*, v. 32, n. 04, p. 620–624, 1993.

BAYER, J. et al. PuLSE: A methodology to develop software product lines. In: *Symposium on Software Reusability (SSR)*. Los Angeles, USA: ACM Press, 1999. p. 122–131.

BAYER, J.; MUTHIG, D.; WIDEN, T. Customizable domain analysis. In: *First International Symposium on Generative and Component-Based Software Engineering (GPCE)*. Germany: [s.n.], 1999. p. 178–194.

BIERHOFF, K.; LIONGOSARI, E. S.; SWAMINATHAN, K. S. Incremental development of a domain-specific language that supports multiple application styles. In: GRAY, J.; TOLVANEN, J.-P.; SPRINKLE, J. (Ed.). *6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon USA*. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, ISBN: 951-39-2631-1, ISSN: 1239-291X, 2006. p. 67–78.

BIGGERSTAFF, T. J.; RICHTER, C. Reusability framework, assessment, and directions. In: BIGGERSTAFF, T. J.; PERLIS, A. J. (Ed.). *Frontier Series: Software Reusability: Volume I - Concepts and Models*. New York: ACM Press, 1989. p. 1–17.

BLOIS, A. P. T. B. *Uma Abordagem de Projeto Arquitetural Baseado em Componentes no Contexto de Engenharia de Domínio*. Tese (Tese de Doutorado) — Universidade Federal do Rio de Janeiro, 2006.

BOEHM, B. A spiral model of software development and enhancement. *IEEE Computer*, v. 21, n. 5, p. 61–72, May 1988.

BOSCH, J. *Design and Use of Software Architectures*. [S.l.]: Addison Wesley, 2000.

BOTTERWECK, G.; O'BRIEN, L.; THIEL, S. Model-driven derivation of product architectures. In: STIREWALT, R. E. K.; EGYED, A.; FISCHER, B. (Ed.). *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. [S.l.]: ACM, 2007. p. 469–472. ISBN 978-1-59593-882-4. Disponível em: <http://doi.acm.org/10.1145/1321631.1321711>. Acesso em 14 jun 2009.

BRAGA, R. T. V. *Um Processo para Construção e Instanciação de Frameworks baseados em uma Linguagem de Padrões para um Domínio Específico*. Tese (Doutorado) — USP - Universidade de São Paulo, São Carlos - SP, 2002.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. [S.l.]: John Wiley & Sons, 2007.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. [S.l.]: John Wiley & Sons, 2007.

BUSCHMANN, F. et al. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. [S.l.]: John Wiley & Sons, 1996.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 20, n. 6, p. 476–493, 1994. ISSN 0098-5589.

CLEAVELAND, J. C. Building application generators. *IEEE Software*, v. 7, n. 1, p. 25–33, 1988.

CLEAVELAND, J. C. Separating concerns of modeling from artifact generation using XML. In: *Proceedings of the 1st OOPSLA Workshop on Domain-Specific Visual Languages - Tampa Bay, USA*. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, ISBN: 951-39-1056-3, ISSN: 0359-8470, 2001. p. 83–86.

CLEENEWERCK, T.; KURTEV, I. Separation of concerns in translational semantics for DSLs in model engineering. In: CHO, Y. et al. (Ed.). *22nd Annual ACM Symposium on Applied Computing (SAC 2007)*. [S.l.]: ACM, 2007. p. 985–992. ISBN 1-59593-480-4.

CLEMENTS, P.; KAZMAN, R.; KLEIN, M. *Evaluating Software Architectures: Methods and Case Studies*. [S.l.]: Addison-Wesley, 2004.

CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. [S.l.]: Addison-Wesley, 2002.

COOK, S. et al. *Domain-Specific Development with Visual Studio DSL Tools*. [S.l.]: Addison-Wesley Professional, 2007.

COPLIEN, J.; HOFFMAN, D.; WEISS, D. Commonality and variability in software engineering. *IEEE Software*, v. 15, n. 6, p. 37–45, 1998.

COPLIEN, J. O. *Multi-Paradigm Design*. Tese (Doutorado) — Vrije Universiteit Brussel, 2000.

COPLIEN, J. O. *A Pattern Definition (http://hillside.net/patterns/definition.html)*. [S.l.]: hillside.net, 2006.

CURRY, W. et al. Empirical analysis of the correlation between amount-of-reuse metrics in the c programming language. In: *SSR '99: Proceedings of the 1999 symposium on Software reusability*. New York, NY, USA: ACM, 1999. p. 135–140. ISBN 1-58113-101-1.

CZARNECKI, K. Overview of generative software development. In: BANÂTRE, J.-P. et al. (Ed.). *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers*. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3566), p. 326–341. ISBN 3-540-27884-2.

CZARNECKI, K. et al. Model-driven software product lines. In: *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*. San Diego, CA, USA: ACM, 2005. p. 126–127.

CZARNECKI, K. et al. Generative programming for embedded software: An industrial experience report. In: BATORY, D.; CONSEL, C.; TAHA, W. (Ed.). *Generative Programming and Component Engineering*. [S.l.]: Springer Berlin / Heidelberg, 2002. (Lecture Notes in Computer Science, v. 2487), p. 156–172.

CZARNECKI, K.; EISENECKER, U. W. *Generative Programming: Methods, Tools, and Applications*. [S.l.]: Addison-Wesley, 2000.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, v. 45, n. 3, p. 621–645, 2006. ISSN 0018-8670. Disponível em: <http://www.research.ibm.com/journal/sj/453/czarnecki.html>. Acesso em: 14 jun 2009.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. *Formalizing Cardinality-based Feature Models and their Staged Configuration*. [S.l.], 2004. Disponível em: <http://www.ece.uwaterloo.ca/ kczarnec/TR04-11.pdf>.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Staged configuration using feature models. In: NORD, R. L. (Ed.). *Proceedings of the Third Software Product Line Conference*. Boston, MA: Springer, 2004. (LNCS 3154), p. 266–283.

DAVIS, T. The reuse capability model: A basis for improving an organization's reuse capability. In: *Proceedings of 2nd ACM/IEEE International Workshop on Software Reusability*. [S.l.]: IEEE Computer Society Press / ACM Press, 1993. p. 126–133.

DEBAUD, J.; SCHMID, K. A systematic approach to derive the scope of software product lines. In: *21st International Conference on Software Engineering (ICSE 99)*. Los Angeles, CA, USA: [s.n.], 1999. p. 34–43.

DEBAUD, J.-M.; FLEGE, O.; KNAUBER, P. PuLSE-DSSA: A method for the development of software reference architectures. In: *ISAW '98: Proceedings of the third international workshop on Software architecture*. New York, NY, USA: ACM, 1998. p. 25–28. ISBN 1-58113-081-3.

DEELSTRA, M. et al. Model driven architecture as approach to manage variability in software product families. In: *Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003)*. Enschede, The Netherlands: [s.n.], 2003. p. 109–114.

DESOUZA, K. C.; AWAZU, Y.; TIWANA, A. Four dynamics for bringing use back into software reuse. *Communications of the ACM*, v. 49, n. 01, p. 97–100, 2006.

DEURSEN, A. v.; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices - ACM Press*, v. 35, n. 6, p. 26–36, 2000.

DEURSEN, A. van; KLINT, P. Little languages: little maintenance. *Journal of Software Maintenance*, John Wiley & Sons, Inc., New York, NY, USA, v. 10, n. 2, p. 75–92, 1998. ISSN 1040-550X.

DEVANBU, P. et al. Analytical and empirical evaluation of software reuse metrics. In: *ICSE '96: Proceedings of the 18th international conference on Software engineering*. Washington, DC, USA: IEEE Computer Society, 1996. p. 189–199. ISBN 0-8186-7246-3.

DOE, D. D.; BERSOFF, E. H. The software productivity consortium (SPC): An industry initiative to improve the productivity and quality of mission-critical software. *Journal of Systems and Software*, v. 6, n. 4, p. 367–378, 1986. Elsevier Science Inc., New York, NY, USA.

D'SOUZA, D.; WILLS, A. *Objects, Components and Frameworks with UML: The Catalysis Approach*. [S.l.]: Addison-Wesley, 1999. (Object Technology Series).

ECKLUND JR, E. F.; DELCAMBRE, L. M. L.; FREILING, M. J. Change cases: Use cases that identify future requirements. In: *Conference Proceedings of the OOPSLA 96, San Jose, CA, USA*. [S.l.]: ACM Press, 1996. p. 342–358.

ECLIPSE. *The Eclipse Modeling Framework (EMF) Overview*. [S.l.]: Eclipse website, 2005.

ECLIPSE. *Eclipse Modeling Framework FAQ*. 2006.

ENDRES, A. Lessons learned in an industrial software lab. *IEEE Software*, v. 10, n. 05, p. 58–61, 1993.

ESSER, R.; JANNECK, J. W. A framework for defining domain-specific visual languages. In: *Proceedings of the 1st OOPSLA Workshop on Domain-Specific Visual Languages - Tampa Bay, USA*. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, ISBN: 951-39-1056-3, ISSN: 0359-8470, 2001. p. 111–124.

EZRAN, M.; MORISIO, M.; TULLY, C. *Practical Software Reuse*. [S.l.]: Springer, 2002.

FEILKAS, M. How to represent models, languages and transformations? In: GRAY, J.; TOLVANEN, J.-P.; SPRINKLE, J. (Ed.). *6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon USA*. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, ISBN: 951-39-2631-1, ISSN: 1239-291X, 2006. p. 169–176.

FENTON, N.; PFLEEGER, S. L. *Software Metrics: A Rigorous and Practical Approach*. 2nd. ed. [S.l.]: Course Technology, 1998.

FLORIJN, G.; MEIJERS, M.; WINSEN, P. v. Tool support for object-oriented patterns. In: *European Conference on Object-Oriented Programming (ECOOP'97)*. Jyväskylä, Finland: Springer-Verlag, 1997. p. 472–495.

FOWLER, M. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. Disponível em: <http://martinfowler.com/articles/injection.html>. Acesso em: 14 jun 2009.

FOWLER, M. *Language Workbenches: The Killer-App for Domain Specific Languages?* [S.l.]: martinfowler.com, 2005. Disponível em: <http://www.martinfowler.com/articles/languageWorkbench.html>. Acesso em: 14 jun 2009.

FOWLER, M. et al. *Refactoring: Improving the Design of Existing Code*. [S.l.]: Addison Wesley, 1999.

FRAKES, W. B.; ISODA, S. Success factors of systematic software reuse. *IEEE Software*, v. 11, n. 01, p. 14–19, 1994.

FRAKES, W. B.; PRIETO-DÍAZ, R.; FOX, C. DARE: Domain Analysis and Reuse Environment. *Annals of Software Engineering*, v. 05, p. 125–141, 1998.

FRAKES, W. B.; TERRY, C. Reuse level metrics. In: *Proceedings of the 3rd IEEE International Conference on Software Reuse (ICSR): Advances in Software Reusability, Rio de Janeiro, Brazil*. [S.l.: s.n.], 1994. p. 139–148.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: *29th International Conference on Software Engineering 2007 - Future of Software Engineering*. Minneapolis, MN, USA: IEEE Computer Society, 2007. p. 37–54.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

GARCIA, V. C. et al. Towards an assessment method for software reuse capability. In: *8th International Conference on Quality Software (QSIC), Oxford, UK*. [S.l.: s.n.], 2008. p. 294–299.

GARCIA, V. C. et al. Towards a maturity model for a reuse incremental adoption. In: *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2007)*. Campinas, SP, Brazil: [s.n.], 2007. p. 61–74.

GENERO, M. et al. Building measure-based prediction models for uml class diagram maintainability. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 5, p. 517–549, 2007. ISSN 1382-3256.

GENERO, M.; POELS, G.; PIATTINI, M. Defining and validating metrics for assessing the understandability of entity-relationship diagrams. *Data Knowl. Eng.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 64, n. 3, p. 534–557, 2008. ISSN 0169-023X.

GREENFIELD, J. et al. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. [S.l.]: Wiley, 2004.

GRISS, M. Software reuse experience at Hewlett-Packard. In: *16th International Conference on Software Engineering (ICSE)*. Sorrento, Italy: IEEE/CS Press, 1994. p. 270.

GRISS, M. Making software reuse work at hewlett-packard. *IEEE Software*, v. 12, n. 01, p. 105–107, 1995.

GRISS, M.; FAVARO, J.; D'ALESSANDRO, M. Integrating feature modeling with the RSEB. In: *The Fifty International Conference on Software Reuse (ICSR)*. Victoria, Canada: IEEE/CS Press, 1998. p. 76–85.

GUERRA, E.; LARA, J. de; DÍAZ, P. Visual specification of measurements and redesigns for domain specific visual languages. *J. Vis. Lang. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 19, n. 3, p. 399–425, 2008. ISSN 1045-926X.

GUIZZARDI, G.; PIRES, L. F.; SINDEREN, M. J. V. On the role of domain ontologies in the design of domain-specific visual modeling languages. In: *In: Proceedings of the 2nd Workshop on Domain-Specific Visual Languages, 17th ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2002)*. [S.l.: s.n.], 2002.

HADDAD, H.; TESSER, H. Reusable subsystems: Domain-based approach. In: *2002 ACM Symposium on Applied Computing (SAC'2002)*. Madrid, Spain: ACM, 2002. p. 971–975.

HEINEMAN, G.; COUNCILL, W. *Component-Based Software Engineering: Putting the Pieces Together*. USA: Addison-Wesley, 2001.

HENRIKSSON, A.; LARSSON, H. *A Definition of Round-Trip Engineering*. [S.l.], 2003. Department of Computer and Information Science, Linköpings Universitet, SE-581 83, Linköping, Sweden.

HESSELLUND, A.; CZARNECKI, K.; WASOWSKI, A. Guided development with multiple domain-specific languages. In: ENGELS, G. et al. (Ed.). *Model Driven Engineering Languages and Systems (MoDELS 2007)*. [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4735), p. 46–60. ISBN 978-3-540-75208-0.

HETTEL, T.; LAWLEY, M. J.; RAYMOND, K. Model synchronisation: Definitions for round-trip engineering. In: VALLECILLO, A.; GRAY, J.; PIERANTONIO, A. (Ed.). *Theory and Practice of Model Transformations (ICMT 2008)*. [S.l.]: Springer Berlin / Heidelberg, 2008. (Lecture Notes in Computer Science, v. 5063/2008), p. 31–45.

HOLIBAUGH, R. et al. Reuse: where to begin and why. In: *TRI-Ada '89: Proceedings of the conference on Tri-Ada '89*. Pittsburgh, Pennsylvania, United States: ACM Press, 1989. p. 266 – 277.

ISAKOWITZ, T.; KAUFFMAN, R. J. Supporting search for reusable software objects. *IEEE Transactions on Software Engineering*, v. 22, n. 6, p. 407–423, 1996.

JACKSON, E. K.; SCHULTE, W. Compositional modeling for data-centric business applications. In: PAUTASSO, C.; TANTER Éric (Ed.). *Software Composition*. [S.l.]: Springer, 2008. (Lecture Notes in Computer Science, v. 4954), p. 190–205. ISBN 978-3-540-78788-4.

JACOBSON, I.; GRISS, M.; JONSSON, P. *Reuse-driven Software Engineering Business (RSEB)*. [S.l.]: Addison-Wesley, 1997.

JACOBSON, I.; LINDSTROM, F. Re-engineering of old systems to an object-oriented architecture. In: *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*. Phoenix, Arizona, United States: ACM Press, 1991. p. 340–350.

JARZABEK, S. Modeling multiple domains in software reuse. In: *The 1997 Symposium on Software Reusability*. Boston, Massachusetts, United States: ACM, 1997. p. 65–74.

JEZEQUEL, J. M.; MEYER, B. Design by contract: The lessons of Ariane. *IEEE Computer*, v. 30, n. 01, p. 129–130, 1997.

JOHNSON, R.; FOOTE, B. Designing reusable classes. *Journal of Object Oriented Programming*, v. 1, n. 2, p. 22–35, 1988.

JOHNSON, R. E. Components, frameworks, patterns. In: *SSR '97: Proceedings of the 1997 symposium on Software reusability*. Boston, Massachusetts, United States: ACM Press, 1997. p. 10–17.

JOHNSON, R. E. Frameworks = (components + patterns). *Communications of the ACM*, v. 40, n. 10, p. 39–42, 1997.

JOOS, R. Software reuse at Motorola. *IEEE Software*, v. 11, n. 05, p. 42–47, 1994.

JOUAULT, F.; BÉZIVIN, J.; KURTEV, I. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: JARZABEK, S.; SCHMIDT, D. C.; VELDHUIZEN, T. L. (Ed.). *Fifth International Conference on Generative Programming and Component Engineering (GPCE'06)*. [S.l.]: ACM, 2006. p. 249–254. ISBN 1-59593-237-2. Disponível em: <http://doi.acm.org/10.1145/1173706.1173744>. Acesso em: 14 jun 2009.

JOUAULT, F.; KURTEV, I. On the architectural alignment of ATL and QVT. In: *ACM Symposium on Applied Computing (SAC 06), model transformation track*. Dijon, Bourgogne, France: [s.n.], 2006. p. 1188–1195.

KANG, K. et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. [S.l.], 1990. Technical Report CMU/SEI-90-TR-21 - Carnegie Mellon University/Software Engineering Institute.

KANG, K. et al. FORM: A Feature-Oriented Reuse Method with domain-specific reference architectures. *Annals of Software Engineering Notes*, v. 05, p. 143–168, 1998.

KANG, K.; LEE, J.; DONOHOE, P. Feature-oriented product line engineering. *IEEE Software*, v. 19, n. 04, p. 58–65, 2002.

KEEPENCE, B.; MANNION, M. Using patterns to model variability in product families. *IEEE Software*, v. 16, n. 4, p. 102–108, 1999.

KETTEMANN, S.; MUTHIG, D.; ANASTASOPOLOUS, M. *Product Line Implementation Technologies - Component Technology View*. [S.l.], March 2003. Fraunhofer IESE Tech Report 015.03/E.

KICZALES, G. et al. Aspect-oriented programming. In: *11st European Conference Object-Oriented Programming (ECOOP'97)*. Finland: Springer Verlag, 1997. (LNCS, v. 1241), p. 220–242.

KIM, M.; YANG, H.; PARK, S. A domain analysis method for software product lines based on scenarios, goals and features. In: *Tenth Asia-Pacific Software Engineering Conference (APSEC)*. Thailand: [s.n.], 2003. p. 126–136.

KIRCHER, M.; JAIN, P. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. [S.l.]: John Wiley & Sons, 2003.

KLEPPE, A.; WARMER, J.; BAST, W. *MDA Explained - The Model Driven Architecture: Practice and Promise*. [S.l.]: Addison-Wesley, 2003. (Object Technology Series).

KLEPPE, A. G. A language description is more than a metamodel. In: *Fourth International Workshop on Software Language Engineering, Nashville, USA*. [S.l.: s.n.], 2007. ISBN not assigned.

KNODEL, J. et al. An efficient migration to model-driven development (MDD). *Electronic Notes in Theoretical Computer Science*, v. 137, n. 3, p. 17–27, 2005.

KOLTUN, P.; HUDSON, A. A reuse maturity model. In: *4th Annual Workshop on Software Reuse*. Hemdon, Virginia: Center for Innovative Technology: [s.n.], 1991.

KOTULA, J. Using patterns to create component documentation. *IEEE Software*, v. 15, n. 2, p. 84–92, 1998.

KRUCHTEN, P. The 4+1 view model of architecture. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 6, p. 42–50, 1995. ISSN 0740-7459.

KRUEGER, C. Software reuse. *ACM Computing Surveys*, v. 24, n. 02, p. 131–183, 1992.

KURTEV, I. et al. Model-based DSL frameworks. In: TARR, P. L.; COOK, W. R. (Ed.). *OOPSLA Companion*. [S.l.]: ACM, 2006. p. 602–616. ISBN 1-59593-491-X. Disponível em: <http://doi.acm.org/10.1145/1176617.1176632>. Acesso em: 14 jun 2009.

LANGE, C.; CHAUDRON, M. An empirical assessment of completeness in uml designs. In: *Proceedings of the 8th Conference on Empirical Assessment in Software Engineering (EASE04)*. [S.l.: s.n.], 2004. p. 111–121.

LANGE, C. F. J.; CHAUDRON, M. R. V. Managing model quality in UML-based software development. In: *STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*. Washington, DC, USA: IEEE Computer Society, 2005. p. 7–16. ISBN 0-7695-2639-X.

LEDECZI, A. et al. Composing domain-specific design environments. *IEEE Computer*, v. 34, n. 11, p. 44–51, 2001.

LEE, J.; KANG, K. C. Feature binding analysis for product line component development. In: LINDEN, F. van der (Ed.). *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*. [S.l.]: Springer, 2003. (Lecture Notes in Computer Science, v. 3014), p. 250–260. ISBN 3-540-21941-2.

LEE, J.; MUTHIG, D. Feature-oriented variability management in product line engineering. *Communications of the ACM*, v. 49, n. 12, p. 55–59, December 2006.

LEE, K.; KANG, K. C. Feature dependency analysis for product line component design. In: *8th International Conference on Software Reuse (ICSR)*. Madrid, Spain: [s.n.], 2004. p. 69–85.

LEE, K.; KANG, K. C.; LEE, J. Concepts and guidelines of feature modeling for product line software engineering. In: *7th International Conference on Software Reuse (ICSR): Methods, Techniques, and Tools*. Austin, Texas: [s.n.], 2002. p. 62–77.

LIM, W. C. Effects of reuse on quality, productivity and economics. *IEEE Software*, v. 11, n. 5, p. 23–30, 1994.

LINDEN, F. V. D.; SCHMID, K.; ROMMES, E. *Software Product Lines In Action: The Best Industrial Practice In Product Line Engineering*. [S.l.]: Springer, 2007.

LISBOA, L. B. *ToolDAy - A Tool for Domain Analysis*. Dissertação (Mestrado) — Federal University of Pernambuco, Recife, Pernambuco, Brazil, 2007.

LUCRÉDIO, D.; ALMEIDA, E. S. d.; PRADO, A. F. d. A survey on software components search and retrieval. In: STEINMETZ, R.; MAUTHE, A. (Ed.). *30th IEEE EUROMICRO Conference, Component-Based Software Engineering Track*. Rennes - France: IEEE/CS Press, 2004. p. 152–159.

LUCRÉDIO, D. et al. Software reuse: The Brazilian industry scenario. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 81, n. 6, p. 996–1013, 2008. ISSN 0164-1212.

LUCRÉDIO, D. et al. The Draco approach revisited: Model-driven software reuse. In: *VI WDBC - Workshop de Desenvolvimento Baseado em Componentes*. Recife - PE - Brazil: [s.n.], 2006. p. 72–79.

LUCRÉDIO, D. et al. Performing domain analysis for model-driven software reuse. In: *10th International Conference on Software Reuse*. Beijing, China: Springer-Verlag, 2008. (Lecture Notes in Computer Science, v. 5030), p. 200–211.

LUCRÉDIO, D.; JACKSON, E. K.; SCHULTE, W. Playing with fire: Harnessing the hottest technologies for ultra-large-scale systems. In: *15th Monterey Workshop - Foundations of Computer Software, Future Trends and Techniques for Development, September 24-26, 2008, Budapest University of Technology and Economics*. [S.l.: s.n.], 2008.

MAIDEN, N.; SUTCLIFFE, A. A computational mechanism for parallel problem decomposition during requirements engineering. In: *8th International Workshop on Software Specification and Design*. Germany: [s.n.], 1996. p. 159–163.

MARTIN, R. *OO Design Quality Metrics: An Analysis of Dependencies*. 1994. Disponível em: <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>. Acesso em: 14 jun 2009.

MASCENA, J. C. C. P. C.R.U.I.S.E - Component Reuse In Software Engineering. In: ____. [S.l.]: C.E.S.A.R. e-books, 2007. cap. 6 - Software Reuse Metrics, p. 125–136.

MASCENA, J. C. C. P.; ALMEIDA, E. S. d.; MEIRA, S. R. d. L. A comparative study on software reuse metrics and economic models from a traceability perspective. In: *IEEE International Conference on Information Reuse and Integration (IRI)*. Las Vegas, Nevada, USA: IEEE/CS Press, 2005. p. 72–77.

MATOS JR, P. O. A. *Analysis of techniques for implementing software product lines variabilities*. Dissertação (M.Sc. Dissertation) — Universidade Federal de Pernambuco - Centro de Informática, Recife, PE, Brazil, August 2008.

MCCABE, T. J. A complexity measure. *IEEE Transactions on Software Engineering*, v. 2, n. 4, p. 308–320, 1976.

MCILROY, M. D. 'Mass produced' software components. In: *NATO Software Engineering Conference*. [S.l.: s.n.], 1968. p. 138–155.

MEI, H.; ZHANG, W.; GU, F. A feature oriented approach to modeling and reusing requirements of software product lines. In: *27th IEEE International Computer Software and Applications Conference (COMPSAC)*. USA: [s.n.], 2003. p. 250–256.

MELLOR, S. J.; CLARK, A. N.; FUTAGAMI, T. Model-driven development. *IEEE Software*, v. 20, n. 5, p. 14–18, 2003.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys*, v. 37, n. 4, p. 316–344, dez. 2005. ISSN 0360-0300.

MILI, H. et al. *Reuse-Based Software Engineering: Techniques, Organization, and Controls*. [S.l.]: John Wiley & Sons, 2002.

MILI, H.; MILI, F.; MILI, A. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, v. 21, n. 6, p. 528–562, 1995.

MODELWARE. *Architecture and Platform Modelling Profiles*. [S.l.], 2006.

MODELWARE. *Definition of Reusable Transformations*. [S.l.], 2006. Disponível em: <http://www.modelware-ist.org/)>. Acesso em: 14 jun 2009.

MODELWARE. *Engineering Metrics Definition*. [S.l.], 2006. Disponível em: <http://www.modelware-ist.org/)>. Acesso em: 14 jun 2009.

MODELWARE. *MDD Maturity Model*. [S.l.], 2006. Disponível em: <http://www.modelware-ist.org/)>. Acesso em: 14 jun 2009.

MODELWARE. *MDD Process Framework*. [S.l.], 2006. Disponível em: <http://www.modelware-ist.org/)>. Acesso em: 14 jun 2009.

MODELWARE. *Model Transformation Tool Suite - Prototype*. [S.l.], 2006. Disponível em: <http://www.modelware-ist.org/)>. Acesso em: 14 jun 2009.

MODELWARE. *ModelBus: Functional & Technical architecture document (Vols I and II)*. [S.l.], 2006. Disponível em: <http://www.modelware-ist.org/)>. Acesso em: 14 jun 2009.

MODELWARE. *Standards Proposals Submitted*. [S.l.], 2006. Disponível em: <http://www.modelware-ist.org/)>. Acesso em: 14 jun 2009.

MOHAGHEGHI, P.; AAGEDAL, J. Evaluating quality in model-driven engineering. In: *MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. p. 6. ISBN 0-7695-2953-4.

MOHAGHEGHI, P.; DEHLEN, V. Where is the proof? - a review of experiences from applying MDE in industry. In: *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 432–443. ISBN 978-3-540-69095-5.

MONPERRUS, M. et al. A model-driven measurement approach. In: *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 505–519. ISBN 978-3-540-87874-2.

MOON, M.; YEOM, K. An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. *IEEE Transactions on Software Engineering*, v. 31, n. 07, p. 551–569, 2005.

MOON, M.; YEOM, K. An approach to developing domain architectures based on variability analysis. In: *Computational Science and Its Applications - ICCSA 2006*. [S.l.]: Springer Berlin / Heidelberg, 2006. (Lecture Notes in Computer Science, v. 3981/2006), p. 441–450.

MOORE, B. et al. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. [S.l.]: ibm.com/redbooks, 2004.

MORILLO, J. L. et al. Incremental software reuse. In: MORISIO, M. (Ed.). *Reuse of Off-the-Shelf Components, 9th International Conference on Software Reuse*. Turin, Italy: Springer, 2006. (Lecture Notes in Computer Science, v. 4039), p. 386–389.

MORISIO, M.; EZRAN, M.; TULLY, C. Success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, v. 28, n. 04, p. 340–357, 2002.

MUSKENS, J.; CHAUDRON, M.; LANGE, C. Investigations in applying metrics to multi-view architecture models. In: *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*. Washington, DC, USA: IEEE Computer Society, 2004. p. 372–379. ISBN 0-7695-2199-1.

MUSZYNSKI, M. Implementing a domain-specific modeling environment for a family of thick-client gui components. In: *The 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego USA*. [S.l.: s.n.], 2005. p. 5–14.

MUTHIG, D. et al. *Technology Dimensions of Product Line Implementation Approaches - State-of-the-art and State-of-the-practice Survey*. [S.l.], September 2002. Tech Report 051.02/E - Fraunhofer IESE.

MUTHIG, D.; PATZKE, T. Generic implementation of product line components. In: *Objects, Components, Architectures, Services and Applications for a Networked World. International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002*. [S.l.]: Springer-Verlag, 2003. (Lecture Notes in Computer Science, v. 2591), p. 313–329.

MUTHIG, D.; PATZKE, T. Implementing software product lines - enhancing reusability by systematically selecting and applying variability mechanisms. In: *Multikonferenz Wirtschaftsinformatik (MKWI) 2004. Band 1 : E-Learning: Modelle, Instrumente und Erfahrungen. Software-Produktlinien. Communities in E-Business*. Berlin: Akademische Verlagsgesellschaft, 2004.

NAUR, P.; RANDELL, B. *Report on NATO Software Engineering Conference*. [S.l.], 1968.

NEIGHBORS, J. M. *Software Construction Using Components*. Tese (Ph.D. Thesis) — University of California at Irvine, 1980.

NEIGHBORS, J. M. Draco: A method for engineering reusable software systems. In: BIGGERSTAFF, T. J.; PERLIS, A. J. (Ed.). *Software Reusability, Volume 1 : Concepts and Models*. [S.l.]: Addison-Wesley, 1989. p. 295–319.

OMG. *MDA Guide Version 1.0.1*. [S.l.], 2003. Disponível em: <http://www.omg.org>. Acesso em: 14 jun 2009.

OMG. *MOF 2.0 Query / Views / Transformations Final Adopted Specification*. [S.l.], 2005. Disponível em: <http://www.omg.org>. Acesso em: 14 jun 2009.

OMG. *Catalog of UML Profiles Specifications*. 2006. Disponível em: <http://www.omg.org>. Acesso em: 14 jun 2009.

OMG. *Meta Object Facility Core Specification Version 2.0*. [S.l.], 2006. Disponível em: <http://www.omg.org>. Acesso em: 14 jun 2009.

OMG. *XML Metadata Interchange (XMI) Specification*. [S.l.], 2006. Disponível em: <http://www.omg.org>. Acesso em: 14 jun 2009.

OMG. *Unified Modeling Language (OMG UML) Infrastructure*. [S.l.], 2007. Disponível em: <http://www.omg.org>. Acesso em: 14 jun 2009.

PARNAS, D. L. On the design and development of program families. *IEEE Transactions on Software Engineering*, v. 2, n. 1, p. 1–9, mar. 1976.

PATZKE, T.; MUTHIG, D. *Product Line Implementation Technologies - Programming Language View*. [S.l.], October 2002. Tech Report 057.02/E - Fraunhofer IESE.

PÉREZ-MARTÍNEZ, J. E.; SIERRA-ALONSO, A. From analysis model to software architecture: A PIM2PIM mapping. In: RENSINK, A.; WARMER, J. (Ed.). *ECMDA-FA*. [S.l.]: Springer, 2006. (Lecture Notes in Computer Science, v. 4066), p. 25–39. ISBN 3-540-35909-5.

PETRE, M. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, ACM, New York, NY, USA, v. 38, n. 6, p. 33–44, 1995. ISSN 0001-0782.

PHOHA, V. A standard for software documentation. *IEEE Computer*, v. 30, n. 10, p. 97–98, 1997.

PILGRIM, J. Measuring the level of abstraction and detail of models in the context of MDD. In: *Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, revised selected papers*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 105–114. ISBN 978-3-540-69069-6.

POPMA, R. *JET Tutorial Part 1 (Introduction to JET)*. May 2004. Eclipse Corner Article. Disponível em: <http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html>. Acesso em: 14 jun 2009.

POULIN, J. Measuring software reusability. In: *Proceedings of the 3rd IEEE International Conference on Software Reuse (ICSR): Advances in Software Reusability, Rio de Janeiro, Brazil*. [S.l.: s.n.], 1994. p. 126–138.

POULIN, J.; CARUSO, J.; HANCOCK, D. The business case for software reuse. *IBM Systems Journal*, v. 32, n. 4, p. 567–594, 1993.

POULIN, J. S.; CARUSO, J. M. A reuse metrics and return on investment model. In: PRIETO-DIAZ, R.; FRAKES, W. B. (Ed.). *Proceedings of 2nd ACM/IEEE International Workshop on Software Reusability*. [S.l.]: IEEE Computer Society Press / ACM Press, 1993. p. 152–167. ISBN 0-8186-3130-9, 0-8186-3132-5, 0-8186-3131-7.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. [S.l.]: McGraw-Hill, 2005.

PRIETO-DÍAZ, R. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, v. 15, n. 2, p. 47–54, 1990.

PRIETO-DÍAZ, R. Making software reuse work: An implementation model. *ACM SIGSOFT Software Engineering Notes*, v. 16, n. 3, p. 61–68, 1991.

PRIETO-DÍAZ, R. Status report: Software reusability. *IEEE Software*, v. 10, n. 3, p. 61–66, 1993. IEEE Computer Society Press. Los Alamitos, CA, USA.

PYSTER, A. B.; BARNES, B. H. The software productivity consortium reuse program. In: *COMPCON'88, Digest of Papers, Thirty-Third IEEE Computer Society International Conference*. San Francisco, California, USA: IEEE Computer Society, 1988. p. 242–249.

RABISER, R.; GRÜNBACHER, P.; DHUNGANA, D. Supporting product derivation by adapting and augmenting variability models. In: *SPLC*. [S.l.]: IEEE Computer Society, 2007. p. 141–150. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.22>. Acesso em: 14 jun 2009.

RIBEIRO, M. de M.; MATOS, J. P.; BORBA, P. A decision model for implementing product lines variabilities. In: *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008. p. 276–277. ISBN 978-1-59593-753-7.

RINE, D. Success factors for software reuse that are applicable across domains and businesses. In: *ACM Symposium on Applied Computing*. San Jose, California, USA: ACM Press, 1997. p. 182–186.

RINE, D.; SONNEMANN, R. Investment in reusable software. a study on software reuse investment success factors. *The Journal of Systems and Software*, v. 41, p. 17–32, 1998.

RINE, D. C.; NADA, N. An empirical study of a software reuse reference model. *Information and Software Technology*, v. 42, n. 1, p. 47–65, 2000.

ROBBES, R.; LANZA, M. Example-based program transformation. In: CZARNECKI, K. et al. (Ed.). *MoDELS*. [S.l.]: Springer, 2008. (Lecture Notes in Computer Science, v. 5301), p. 174–188. ISBN 978-3-540-87874-2.

ROMBACH, D. Fraunhofer: The german model for applied research and technology transfer. In: *22nd International Conference on Software Engineering (ICSE)*. Limerick, Ireland: ACM Press, 2000. p. 25–34.

ROTHENBERGER, M. A. et al. Strategies for software reuse: A principal component analysis of reuse practices. *IEEE Transactions on Software Engineering*, v. 29, n. 09, p. 825–837, 2003.

SAMETINGER, J. *Software Engineering with Reusable Components*. Berlin Heidelberg: Springer-Verlag, 1997.

SANTOS, A. L.; KOSKIMIES, K.; LOPES, A. Automated domain-specific modeling languages for generating framework-based applications. In: *SPLC*. [S.l.]: IEEE Computer Society, 2008. p. 149–158. ISBN 978-0-7695-3303-2. Disponível em: <http://dx.doi.org/10.1109/SPLC.2008.17>. Acesso em 14 jun 2009.

SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, v. 39, n. 2, p. 25–31, 2006.

SCHMIDT, D. C. et al. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. [S.l.]: John Wiley & Sons, 2000.

SEACORD, R.; PLAKOSH, D.; LEWIS, A. *Modernizing Legacy Systems - Software Technologies, Engineering Processes, and Business Practices*. [S.l.]: Addison-Wesley, 2003.

SHERIF, K.; APPAN, R.; LIN, Z. Resources and incentives for the adoption of systematic software reuse. *International Journal of Information Management*, v. 26, n. 1, p. 70–80, 2006. Available online 10 October 2005.

SILVA, M. F.; WERNER, C. M. L. Packaging reusable components using patterns and hypermedia. In: *4th International Conference on Software Reuse*. USA: [s.n.], 1996. p. 146–155.

SIMOS, M. et al. *Organization Domain Modeling (ODM) Guidebook Version 2.0*. [S.l.], 1996. STARS Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, Manassas VA.

SOMMERVILLE, I. *Software Engineering*. 8. ed. [S.l.]: Addison Wesley, 2006.

STARS. *Software Technology for Adaptable Reliable Systems (STARS). The Reuse-Oriented Software Evolution (ROSE)*. [S.l.], 1993. Unisys STARS Technical Report,Advanced Research Projects Agency (ARPA) STARS Technology Center, 801 N. Randolph St. Suite 400, Arlington VA 22203.

STEFFEN, B. et al. Model-Driven Development with the jABC. In: *Hardware and Software, Verification and Testing*. Berlin / Heidelberg: Springer-Verlag, 2007. (Lecture Notes in Computer Science, v. 4383/2007), p. 92–108.

SVAHNBERG, M.; GURP, J. van; BOSCH, J. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 35, n. 8, p. 705–754, 2005. ISSN 0038-0644.

SVANHBERG, M.; GURP, J. van; BOSCH, J. *A Taxonomy of Variability Realization Techniques*. [S.l.], 2002. Technical paper ISSN:1103-1581, Blekinge Institute of Technology, Sweden, 2002.

SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. [S.l.]: Addison Wesley, 1999.

SZYPERSKI, C.; GRUNTZ, D.; MURER, S. *Component Software: Beyond Object-Oriented Programming - Second Edition*. [S.l.]: Addison Wesley / ACM Press, 2002.

TAULAVUORI, A.; NIEMELA, E.; KALLIO, P. Component documentation-a key issue in software product lines. *Journal of Information and Software Technology*, v. 46, n. 8, p. 535–546, 2004.

THOMAS, D. MDA: Revenge of the modelers or uml utopia? *IEEE Software*, v. 21, n. 3, p. 15–17, 2004.

TOLVANEN, J.-P. Making model-based code generation work. *Embedded Systems Europe*, v. 8, n. 60, p. 36–38, Aug/Sept 2004.

TOLVANEN, J.-P. *Domain-Specific Modeling: How to Start Defining Your Own Language*. Feb 2005. Disponível em: <http://www.devx.com/enterprise/Article/30550>. Acesso em: 14 jun 2009.

TOLVANEN, J.-P.; KELLY, S. Modelling languages for product families: A method engineering approach. In: *Proceedings of the 1st OOPSLA Workshop on Domain-Specific Visual Languages - Tampa Bay, USA*. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, ISBN: 951-39-1056-3, ISSN: 0359-8470, 2001. p. 135–140.

TOLVANEN, J.-P.; KELLY, S. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In: OBBINK, J. H.; POHL, K. (Ed.). *9th International Software Product Line Conference (SPLC-Europe 2005)*. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3714), p. 198–209. ISBN 3-540-28936-4.

TRACZ, W. Domain-Specific Software Architecture (DSSA) pedagogical example. *ACM SIGSOFT Software Engineering Notes*, v. 20, n. 3, p. 49–62, 1995.

TRASK, B. et al. Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications. In: *SPLC*. [S.l.]: IEEE Computer Society, 2006. p. 192–202. ISBN 0-7695-2599-7.

UHL, A. Model Driven Architecture is ready for prime time. *IEEE Software*, v. 20, n. 5, p. 70–71, 2003.

VANDOREN, E. *Maintainability Index Technique for Measuring Program Maintainability*. [S.l.]: Software Engineering Institute (SEI), 1997. Disponível em: <http://www.sei.cmu.edu/str/>. Acesso em: 14 jun 2009.

VARRÓ, D. Model transformation by example. In: NIERSTRASZ, O. et al. (Ed.). *MoDELS*. [S.l.]: Springer, 2006. (Lecture Notes in Computer Science, v. 4199), p. 410–424. ISBN 3-540-45772-0.

VISSER, E. WebDSL: A case study in domain-specific language engineering. In: LÄMMEL, R.; VISSER, J.; SARAIVA, J. (Ed.). *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*. [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 5235), p. 291–373. ISBN 978-3-540-88642-6.

VÖLTER, M. A catalog of patterns for program generation. In: *Eight European Conference on Pattern Languages of Programs (EuroPLoP 2003), Irsee, Germany*. [S.l.: s.n.], 2003. p. B6–1–B6–34.

VÖLTER, M. *MD* Best Practices*. December 2008. Disponível em: <http://www.voelter.de>. Acesso em: 14 jun 2009.

VÖLTER, M.; BETTIN, J. Patterns for model-driven software development. In: *Ninth European Conference on Pattern Languages of Programs (EuroPLoP 2004), Irsee, Germany*. [S.l.: s.n.], 2004. p. D3–1–D3–54.

VÖLTER, M.; GROHER, I. Product line implementation using aspect-oriented and model-driven software development. In: *SPLC*. [S.l.]: IEEE Computer Society, 2007. p. 233–242. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.23>. Acesso em: 14 jun 2009.

VOTH, D. Packaging reusable software assets. *IEEE Software*, v. 21, n. 3, p. 107–110, 2004.

W3C. *XML Path Language (XPath) Version 1.0 - W3C Recommendation 16 November 1999*. [S.l.], 1999.

W3C. *XSL Transformations (XSLT) Version 1.0 - W3C Recommendation 16 November 1999*. [S.l.], 1999.

WARMER, J.; KLEPPE, A. Building a flexible software factory using partial domain specific models. In: *Proc. of The 6th OOPSLA Workshop on Domain-Specific Modeling*. [S.l.: s.n.], 2006. p. 15–22.

WEIS, T.; ULBRICH, A.; GEIHS, K. Model metamorphosis. *IEEE Software*, v. 20, n. 5, p. 46–51, 2003.

WEISS, D.; LAI, C. T. R. *Software Product-Line Engineering: A Family-Based Software Development Process*. [S.l.]: Addison Wesley, 1999.

WEISS, D. M. et al. Decision-model-based code generation for SPLE. In: *SPLC*. [S.l.]: IEEE Computer Society, 2008. p. 129–138. ISBN 978-0-7695-3303-2. Disponível em: <http://dx.doi.org/10.1109/SPLC.2008.42>. Acesso em: 14 jun 2009.

WILE, D. Lessons learned from real DSL experiments. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 51, n. 3, p. 265–290, 2004. ISSN 0167-6423.

WIMMER, M. et al. Towards model transformation generation by-example. In: *40th Hawaii International Conference on System Sciences (HICSS'07)*. Hawaii: [s.n.], 2007. p. 285–294.

WOHLIN, C. et al. *Experimentation in Software Engineering: An Introduction*. [S.l.]: Kluwer Academic Publishers, 2000.

ZAREMSKI, A. M.; WING, J. M. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, v. 4, n. 2, p. 146–170, 1995.

APÊNDICE A – Técnicas para reutilização de software

O método mais simples de reutilização é conhecido e utilizado por praticamente todo desenvolvedor, sob o nome informal de "*copiar e colar*". A função que permite realizar a cópia ou duplicação do objeto de trabalho está presente em quase todas as aplicações e sistemas operacionais existentes na atualidade. Por exemplo, é possível copiar texto de um local para outro, permitindo assim que um desenvolvedor reutilize trechos de código. Outras ferramentas de auxílio à Engenharia de Software, tais como as ferramentas de modelagem, também permitem que sejam duplicados desenhos e outros elementos de modelagem.

Esse tipo de reutilização acontece quando um desenvolvedor está trabalhando em alguma atividade do processo de software, e se depara com uma situação na qual ele se lembra de um artefato (código ou não), que ele já construiu, utilizou, ou que de alguma maneira saiba existir. Ele então localiza este artefato, identifica as partes que lhe serão úteis, e copia para o local onde está trabalhando, realizando as modificações necessárias para integrá-lo ao novo contexto.

Analisando-se esse processo cotidiano, pode-se identificar os quatro conceitos citados por Krueger (1992):

Abstração: a abstração existe na mente do desenvolvedor, de maneira informal, e é formada durante a sua experiência pessoal como Engenheiro de Software. Ele não se lembra exatamente de todos os detalhes dos artefatos com os quais já se deparou. Porém, ele é capaz de abstrair os detalhes, relevando somente o que é essencial, formando uma espécie de biblioteca reutilizável indexada em sua memória;

Seleção: da mesma forma com que consegue abstrair os detalhes, o desenvolvedor também é capaz de utilizar a sua memória seletiva para determinar se o artefato satisfaz suas necessidades, e fornecer pistas de onde encontrá-lo. Este momento corresponde à seleção, quando o desenvolvedor, utilizando essas pistas, se lembra, primeiro, do software ou biblioteca onde o artefato se encontra localizado. Em seguida, ele percorre a estrutura e

documentação do software, navegando, por exemplo, por meio da divisão em pacotes ou bibliotecas, em busca do artefato lembrado por sua memória;

Adaptação: encontrado, o artefato é copiado para seu novo local. A adaptação ocorre somente se necessário, configurando-o por meio de parâmetros ou de mecanismos de extensão ou mesmo modificando seu código manualmente; e

Integração: a integração normalmente consiste em modificar o artefato, o contexto, ou ambos, para que possam compor a funcionalidade desejada (KRUEGER, 1992).

Essa abordagem, apesar de proporcionar ganhos de produtividade, apresenta alguns problemas, tais como:

- Uma vez que o artefato normalmente é modificado, é necessário testá-lo novamente para garantir que as modificações não introduziram nenhum erro novo;
- Os processos de abstração e seleção dependem muito do talento individual e da experiência do desenvolvedor, pois se baseiam muito na memória humana; e
- Caso o desenvolvedor não conheça o artefato sendo reutilizado, ele precisa gastar algum tempo compreendendo-o antes de copiá-lo para o novo contexto. Desta forma, caso esse tempo seja muito grande, o que ocorre na maioria dos casos envolvendo código-fonte é que ele acaba optando por desenvolver um novo artefato a partir do zero (DESOUZA; AWAZU; TIWANA, 2006).

Por esses e outros motivos, a pesquisa vem evoluindo na busca por novos métodos e técnicas que evitem tais tipos de problemas. A seguir, são apresentadas as principais abordagens voltadas para reutilização de software existentes na literatura, destacando-se o desenvolvimento baseado em componentes (SAMETINGER, 1997; SZYPERSKI; GRUNTZ; MURER, 2002), linguagens específicas de domínio (DEURSEN; KLINT; VISSER, 2000), programação generativa (CZARNECKI; EISENECKER, 2000), e outras técnicas, tais como frameworks (JOHNSON, 1997b), padrões (BUSCHMANN et al., 1996) e reengenharia de software (JACOBSON; LINDSTROM, 1991).

Desenvolvimento Baseado em Componentes

Dentre as técnicas utilizadas para se alcançar os benefícios da reutilização, destaca-se o desenvolvimento baseado em componentes (DBC). Até alguns anos atrás, o desenvolvimento da maioria dos produtos de software era baseado em blocos monolíticos, formados por um

grande número de partes inter-relacionadas e, na maioria das vezes, essa relação entre as partes era implícita. O DBC surgiu como uma nova perspectiva para desenvolvimento de software, na qual a analogia é de quebrar esses blocos monolíticos em componentes intercambiáveis, diminuindo a complexidade e explicitando a relação entre as partes que compõem o software.

A idéia de componentes de software não é nova. Em 1968, durante uma conferência da OTAN sobre Engenharia de Software, o foco era a crise do software – o problema de se construir sistemas de software grandes e confiáveis de uma maneira controlável e eficiente. Desde o início, a reutilização foi considerada como uma maneira de solucionar a crise do software. Um artigo apresentado na conferência, intitulado *"Componentes de Software Produzidos em Massa"*, por McIlroy (MCILROY, 1968), se tornou um dos mais revolucionários da área de reutilização. Nas palavras de McIlroy: *"a indústria de software é fracamente fundada e um dos motivos dessa fraqueza é a ausência de uma indústria de sub-componentes"*.

Apesar de serem antigas, as idéias de McIlroy ainda não foram concretizadas. O próprio conceito de componentes ainda não é bem definido, sendo que desde o primeiro workshop de DBC, em 1998 em Kyoto, várias definições foram apresentadas, cada uma evidenciando um aspecto específico do componente. Um conceito bastante satisfatório é apresentado por Szyperski (1999): *"um componente de software é uma unidade de composição com* ***interfaces bem especificadas em contrato*** *e* ***dependências explícitas do contexto****. Um componente de software pode ser* ***independentemente implantado*** *e pode ser* ***composto por terceiros****"*.

As interfaces bem especificadas são importantes para formar uma camada comum entre o reutilizador (uma pessoa que utiliza o componente) e o desenvolvedor do componente. As dependências explícitas definem o que o ambiente deve fornecer para que o componente possa executar sua função. Uma vez satisfeitas estas dependências explícitas, o componente deve poder ser implantado sem a necessidade de resolver dependências adicionais. Finalmente, para que ele possa ser composto por terceiros, ele deve ser suficientemente auto-contido, ou seja, a função por ele desempenhada deve ser totalmente implementada dentro dele.

Um componente não é totalmente independente dos outros componentes e do ambiente. As suas interfaces são importantes justamente para explicitar quais são esses pontos de dependência. Szyperski (1999) define uma interface como um conjunto de assinaturas de operações que podem ser chamadas por uma aplicação. De acordo com Heineman e Councill (2001), há dois tipos de interfaces: interfaces providas, que descrevem as funcionalidades que o componente fornece, e as interfaces requeridas, que descrevem as funcionalidades das quais o componente depende para funcionar.

Os quatro conceitos identificados por Krueger (1992) também estão presentes nessa

abordagem, conforme descritos a seguir:

Abstração: a abstração é alcançada por meio do encapsulamento das funções desempenhadas pelo componente, de forma que o desenvolvedor não tome conhecimento dos detalhes de implementação. Em outras palavras, a **parte visível** corresponde à interface do componente, enquanto que a **parte escondida** corresponde à sua implementação. Para se conseguir essa separação, podem ser utilizados, por exemplo, os conceitos da orientação a objetos, como herança e polimorfismo. Também podem ser criadas descrições em linguagem natural que permitam ao desenvolvedor identificar as funções desempenhadas pelo componente sem a necessidade de conhecer sua estrutura interna;

Seleção: para o processo de seleção, normalmente os catálogos de componentes dispõem de uma estrutura organizada de forma a facilitar a sua navegação. Também é comum o uso de mecanismos automáticos de busca, que reduzem o tempo necessário para que o desenvolvedor encontre aquilo que está procurando. Existe uma vasta literatura em torno desse assunto (LUCRÉDIO; ALMEIDA; PRADO, 2004);

Adaptação: a adaptação do componente é normalmente feita por meio de sua parametrização, na qual o reutilizador especifica parâmetros para que o componente possa executar sua função no contexto em que está sendo reutilizado. Porém, normalmente a parametrização só é possível nos casos previstos durante o projeto do componente. Adaptar um componente para que ele execute em um cenário imprevisto pode exigir que o mesmo seja modificado internamente. Neste caso, se o esforço necessário para essa modificação for muito grande, pode ser mais vantajoso tentar renegociar com o cliente os requisitos da aplicação do que modificar o próprio componente; e

Integração: a integração consiste no acoplamento das interfaces requeridas pelo componente com as interfaces providas pelo restante da aplicação. Normalmente esse acoplamento exige alguma modificação na aplicação.

Diferentemente da abordagem "*copiar e colar*", um componente, ao ser reutilizado, não precisa ser novamente testado caso não tenha sido modificado. Além disso, a existência de uma interface bem definida, assim como os mecanismos de classificação e busca, fazem com que o desenvolvedor não precise confiar tanto em sua memória para encontrar um componente candidato à reutilização.

Linguagens específicas de domínio

O termo Linguagem Específica de Domínio (DSL – *Domain-Specific Language*) exige atenção especial, pois se baseia em uma noção muito vaga, que é a palavra "Domínio", utilizada com diferentes significados em diferentes áreas. Para esclarecer o conceito de domínio considerado nesta pesquisa, a Figura 45 apresenta um exemplo da situação clássica vivida durante o processo de análise de sistemas.

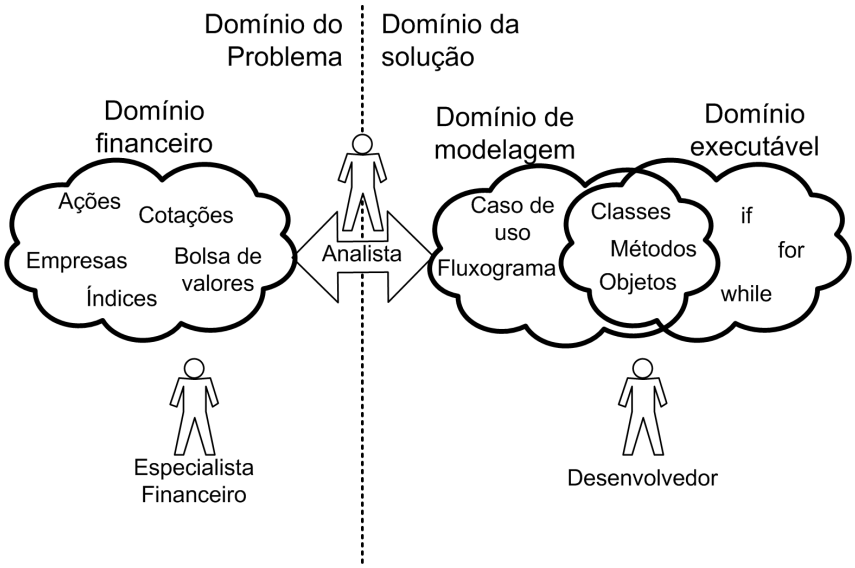


Figura 45: Situação clássica da análise de sistemas

O desenvolvimento de sistemas envolve normalmente a captura do conhecimento de um especialista de alguma área (no exemplo, um especialista financeiro) e sua representação em uma forma que facilite o desenvolvimento de uma solução computacional. É papel do analista de sistemas traduzir os termos e conceitos familiares ao especialista para termos e conceitos de computação, familiares ao desenvolvedor.

Neste contexto, a palavra domínio refere-se a uma determinada área de competência e conhecimento, que possui terminologia e conceitos particulares. No exemplo, os termos e conceitos financeiros, isto é, "Ações", "Índices" e "'Cotações", estão dentro da área de competência e conhecimento do especialista financeiro. Este é, portanto, o **domínio financeiro**.

Os termos e conceitos do desenvolvimento de software, isto é, "Casos de uso", "Classes", "Objetos", "Métodos", as palavras-chave "if", "while", entre outras, estão dentro da área de competência e conhecimento do desenvolvedor. Alguns desses termos fazem parte do **domínio de modelagem**. Outros fazem parte do **domínio executável**, enquanto outros fazem parte de ambos.

Uma outra possível distinção é feita no momento em que o analista realiza essa tradução

dos termos do domínio financeiro para os termos dos domínios de modelagem e executável. No contexto de desenvolvimento de sistemas, o domínio para o qual se está construindo um sistema é chamado de **domínio do problema**, pois trata-se do problema que se pretende resolver. O domínio no qual se está construindo o sistema é chamado de **domínio da solução**, pois ele representa a solução computacional que está sendo desenvolvida.

Em reutilização de software, a palavra domínio é normalmente utilizada como um sinônimo de **Domínio do Problema**. Ou seja, quando se fala em aplicações de um domínio, ou um domínio de aplicações, se fala em um conjunto de aplicações que compartilham os mesmos termos e conceitos, e que são específicas para uma mesma área de conhecimento. Assim, o domínio financeiro, por exemplo, no contexto da reutilização de software, compreende todas as aplicações financeiras.

A partir dessa definição, uma linguagem específica de domínio é uma linguagem qualquer que representa os termos e conceitos do domínio. Por exemplo, Java é uma linguagem de um domínio executável, UML é uma linguagem de um domínio de modelagem, e assim por diante.

Mas atualmente, quando se fala em linguagens específicas de um domínio, normalmente se está querendo dizer linguagens específicas de um **domínio de problema**. Dessa forma, chega-se à seguinte definição (DEURSEN; KLINT; VISSER, 2000).

"Uma linguagem específica de domínio é uma linguagem de programação ou uma linguagem de especificação executável que oferece, por meio de notações e abstrações apropriadas, poder expressivo focado em, e normalmente restrito a, um domínio de um problema particular".

De acordo com essa definição, uma linguagem do domínio financeiro é tida como uma DSL, enquanto Java e UML não o são.

Uma DSL não é necessariamente utilizada com objetivo de gerar um programa executável (FOWLER, 2005). Em muitos casos, pode-se utilizar uma DSL com fins de configuração, por exemplo. Um arquivo XML de configuração de acesso a banco pode ser visto como uma especificação em uma DSL, pois ele contém termos e conceitos associados ao problema de acesso a banco de dados. Outro exemplo são os processadores de texto do tipo LaTeX, que usam uma linguagem própria com o propósito de produzir documentos formatados.

Porém, no caso da reutilização de software, DSLs normalmente são utilizadas com o propósito de se produzir um programa executável (WARMER; KLEPPE, 2006), reaproveitando o conhecimento específico daquele domínio. A seguir discute-se esse tipo de uso para DSLs.

Linguagens específicas de domínio são mais uma expressão da dicotomia entre abordagens

genéricas (que oferecem soluções para múltiplas áreas, porém de forma não otimizada), e abordagens específicas (que oferecem soluções melhores, porém apenas para um subconjunto de problemas), presente na maioria dos ramos da ciência e da engenharia (DEURSEN; KLINT; VISSER, 2000).

As atuais linguagens de programação, como Java, C++, e C#, são exemplos de linguagens de propósito genérico, ou seja, ao criar sistemas para diferentes domínios de problema, pode-se utilizar a mesma linguagem. O mesmo acontece para linguagens de modelagem, como a UML, por exemplo, que pode ser utilizada em diferentes cenários.

No entanto, essas linguagens apresentam o problema de exigirem um certo esforço de tradução para expressar soluções ou modelos específicos para um determinado problema utilizando construções genéricas. Além disso, esse esforço de tradução precisa ser praticamente repetido toda vez que se for construir uma aplicação diferente para aquele mesmo domínio. Para resolver esse problema, três abordagens têm sido adotadas (DEURSEN; KLINT; VISSER, 2000):

1. *Bibliotecas de subrotinas*: consistem em subrotinas que realizam tarefas específicas para um domínio de problema, de forma que o desenvolvedor, ao reutilizar essas subrotinas, também reutiliza conhecimento específico para esse domínio;

2. *Frameworks orientados a objetos e frameworks de componentes*: estendem a idéia de bibliotecas de subrotinas. Enquanto bibliotecas possuem uma estrutura simples, com as subrotinas sendo chamadas pela aplicação, os *frameworks* normalmente estão no controle, sendo os responsáveis por chamar o código específico da aplicação; e

3. *Linguagens específicas de domínio (DSL)*: são linguagens pequenas, normalmente declarativas, com poder expressivo focado em um domínio de problema. Normalmente, programas em DSL são convertidos para programas em linguagens comuns, utilizando *frameworks* ou bibliotecas de subrotinas. Dessa forma, pode-se pensar em uma DSL como uma maneira de esconder os detalhes da biblioteca ou *framework*.

Do ponto de vista da reutilização, todas essas abordagens são similares, apresentando um único propósito: reutilizar o conhecimento do domínio na construção de aplicações daquele domínio. Seja na forma de chamadas de rotinas de uma biblioteca ou utilizando uma linguagem, o resultado final é praticamente o mesmo, com diferenças apenas na forma de expressão da solução. De fato, Martin Fowler, um conceituado pesquisador na área de reutilização, destaca que sempre considerou o fato de definir funções e bibliotecas como uma forma de se construir uma DSL para um problema (FOWLER, 2005).

Independentemente da abordagem adotada, a principal característica de uma DSL é o fato de ela ter seu poder expressivo focado em um domínio do problema, o que reduz o esforço de tradução necessário para construir programas, pois os termos da linguagem estão próximos dos termos reais conhecidos pelo especialista daquele domínio. Portanto, DSLs são normalmente pequenas, consistindo de um conjunto de abstrações e notações restritos a um domínio, de forma que especialistas daquele domínio possam trabalhar nessa linguagem facilmente. Por este motivo, são também conhecidas como micro-linguagens ou linguagens pequenas, principalmente pelos usuários do Sistema Operacional Unix (FOWLER, 2005).

A utilização de DSLs apresenta vantagens e desvantagens. Dentre as vantagens, destacam-se (DEURSEN; KLINT; VISSER, 2000):

- DSLs permitem que soluções sejam expressas nos termos e no nível de abstração do domínio do problema. Consequentemente, especialistas do domínio podem compreender, validar, modificar, ou mesmo desenvolver seus próprios programas;
- Programas DSLs são concisos, auto-documentados, e podem ser reutilizados com diferentes propósitos;
- DSLs aumentam a produtividade, confiabilidade, manutenibilidade e portabilidade;
- DSLs incorporam conhecimento sobre o domínio, e portanto possibilitam sua conservação e reutilização;
- DSLs possibilitam validação e otimização no nível do domínio; e
- DSLs facilitam a testabilidade das aplicações.

As desvantagens de se utilizar DSL são (DEURSEN; KLINT; VISSER, 2000):

- O custo para se projetar, implementar e manter uma DSL;
- O custo de treinamento para usuários da DSL;
- A pouca disponibilidade de DSLs;
- A dificuldade de se definir um escopo adequado para uma DSL;
- A dificuldade de se balancear entre especificidade ao domínio e linguagens de programação genéricas; e

- Para os casos de DSLs executáveis, a perda potencial de desempenho quando comparado com código construído à mão.

Alguns dos benefícios das DSLs, como aumento da produtividade, confiabilidade, manutenibilidade, portabilidade, a reutilização de conhecimento sobre o domínio, entre outros, podem ser igualmente alcançados por meio da utilização de *frameworks*, descritos mais adiante neste apêndice, ou outras técnicas de reutilização. Dessa forma, frente às suas desvantagens, pode-se argumentar que em alguns casos o uso de DSLs é desnecessário. Já outros benefícios, como a possibilidade de se trabalhar nos termos e no nível de abstração do domínio do problema, só podem ser obtidos por meio de DSLs. Neste sentido, a decisão de se utilizar ou não essa técnica se resume à análise dos benefícios obtidos versus o custo necessário para construir as ferramentas necessárias para oferecer um suporte eficiente para DSLs (FOWLER, 2005).

Esse custo depende da forma escolhida para se implementar o suporte para a DSL. Após a análise do domínio e projeto da DSL, existem duas alternativas principais para se implementar esse suporte:

1. **Compilador/interpretador**: a forma mais comum de se implementar uma DSL envolve construir uma biblioteca que implementa as noções semânticas, e projetar e implementar um compilador que traduza programas na DSL para chamadas a essa biblioteca (DEURSEN; KLINT; VISSER, 2000). Esse tipo de abordagem é também conhecido como **DSL externa** (FOWLER, 2005); e

2. **Linguagem embutida**: nesse tipo de DSL, também conhecida como **DSL interna** (FOWLER, 2005), uma linguagem de programação genérica é estendida com conceitos e operações do domínio. A principal vantagem é que o próprio compilador da linguagem base pode ser utilizado. Em contrapartida, a expressividade da linguagem estendida é limitada ao poder expressivo da linguagem base (DEURSEN; KLINT; VISSER, 2000). Por exemplo, a UML (OMG, 2007), com seu mecanismo de extensão, possibilita a criação de linguagens de modelagens específicas de forma bastante razoável. Os perfis UML (OMG, 2006a) são um exemplo desse poder. A linguagem LISP, ou outras linguagens adaptáveis, também são exemplos dessa possibilidade. Já a linguagem Java não possui um mecanismo simples de extensão, dificultando o uso de linguagens embutidas.

Enquanto a primeira abordagem resulta em uma DSL mais limpa e fácil de ser utilizada, ela requer maior trabalho para implementar o compilador. Já a segunda abordagem é mais rápida e simples de ser implementada, pois não requer a construção de um compilador. Porém, os resultados não são tão satisfatórios como na primeira abordagem.

Com relação a linguagens visuais, até pouco tempo o uso do mecanismo de extensão da UML era uma das únicas possibilidades. Recentemente, com as idéias do desenvolvimento orientado a modelos, novas tecnologias surgiram para facilitar esse trabalho. São os chamados *frameworks* de linguagens, que implementam diversas funções necessárias para a modelagem visual, como a representação, criação e edição de diagramas, e o processamento automático de suas informações internas. Exemplos incluem o GME (*Generic Modeling Environment*)[1] e o GMF (*Graphical Modeling Framework*)[2].

Analisando a abordagem de DSLs como uma forma de reutilização de software, pode-se notar que os quatro conceitos da reutilização também estão presentes:

Abstração: o processo de análise de um domínio de problema, levantamento de informações relacionadas, e seu encapsulamento em forma de uma linguagem e uma biblioteca contendo as noções semânticas, corresponde à abstração. O conhecimento reutilizável do domínio é abstraído e expresso em uma linguagem, de forma que um desenvolvedor pode facilmente reconhecê-lo e reutilizá-lo. Essa é a **parte visível** da abstração. O detalhamento e a implementação das noções semânticas associadas a cada conceito correspondem à **parte escondida** da abstração;

Seleção: uma vez que a DSL esteja construída, o desenvolvedor precisa conhecer a sintaxe e a semântica por trás de cada construção da linguagem. A seleção dos artefatos reutilizáveis é então feita pelo desenvolvedor, que escolhe mentalmente os termos e conceitos da linguagem a serem utilizados durante a construção de um programa ou criação de um modelo. Ele pode também consultar a gramática ou manual da linguagem, para ajudá-lo a determinar o que precisa utilizar;

Adaptação: a adaptação ocorre normalmente por meio de parâmetros definidos na própria linguagem, com os quais o desenvolvedor adiciona detalhes específicos da situação; e

Integração: por fim, a integração é normalmente realizada de forma automática por um compilador ou interpretador, que irá executar as ações semânticas associadas aos termos da DSL, integrando os elementos sendo reutilizados em um aplicativo executável. Quando o aplicativo é gerado a partir de um compilador DSL, pode-se também dizer que se trata de um gerador de aplicações, assunto da próxima seção.

[1] `http://www.isis.vanderbilt.edu/projects/gme/`
[2] `http://www.eclipse.org/gmf/`

Programação generativa

"Programação generativa (generative programming) é a automação da manufatura de produtos intermediários e finais (i.e., componentes e aplicações.") (CZARNECKI; EISENECKER, 2000).

Essa definição resume o conceito de programação generativa, que vem sendo utilizado desde os primórdios da computação. Significa que parte ou a totalidade dos artefatos produzidos durante o ciclo de vida do software é gerada automaticamente. Por exemplo, compiladores que utilizam como entrada um programa em uma linguagem de programação, e geram como saída o código executável para uma plataforma, estão entre os primeiros exemplos de uso da programação generativa.

Outro exemplo são os construtores de interface, tais como aqueles presentes em softwares como Microsoft Visual Studio[3] e NetBeans[4]. O desenvolvedor especifica a interface visualmente, e todo o código que a implementa é automaticamente gerado. Caso precise realizar alguma modificação, o desenvolvedor a realiza diretamente no editor visual, e o código que reflete essa mudança é gerado novamente.

A programação generativa pode ser utilizada em outras etapas do ciclo de vida do software. Pode-se gerar casos de teste, esquemas de banco de dados, ou mesmo programas completos. Os benefícios dessa abordagem são óbvios: poupa-se o tempo gasto em atividades repetitivas, como tarefas de implementação de infraestrutura, aproveitando-o em atividades mais importantes, como análise e projeto.

Além de proporcionar os ganhos diretos em termos de tempo de desenvolvimento, a abordagem generativa também promove a reutilização. Um gerador encapsula o conhecimento do domínio necessário para que sejam produzidos artefatos ou mesmo aplicações completas daquele domínio. Em outras palavras, a reutilização desse conhecimento ocorre quando se reutiliza o **processo de geração**, e não os componentes (SAMETINGER, 1997).

Um elemento necessário para a abordagem generativa é o formato da entrada fornecida a um gerador. Normalmente, utiliza-se uma DSL, ou seja, uma linguagem especializada e orientada ao problema (CZARNECKI; EISENECKER, 2000). Desta forma, o desenvolvedor pode criar as especificações de forma mais próxima ao problema, o que exige um menor esforço de especificação.

Essa DSL pode ser tão especializada quanto o necessário para o problema sendo modelado.

[3] `http://msdn.microsoft.com/vstudio/`
[4] `http://www.netbeans.org/`

Por exemplo, para modelar um produto matemático, a DSL precisa ser bem específica, com elementos formais que permitam representar conceitos matemáticos. Outro exemplo é o caso de um editor de interfaces, onde a DSL é visual, sendo composta dos elementos básicos de uma interface, como botões, caixas de texto, barras de rolagem, entre outros.

Pode-se também utilizar *templates* como entrada para um gerador. Um *template* consiste em uma estrutura pré-definida que representa o artefato final semi-concluído, com pontos em aberto que podem ser preenchidos através de variáveis especificadas pelo desenvolvedor.

Um dos problemas da programação generativa ocorre quando se deseja modificar os artefatos gerados. Por mais genérico e flexível que seja o gerador, o artefato gerado pode não corresponder exatamente às necessidades do desenvolvedor. Neste caso, o desenvolvedor pode modificar o gerador, para atender às suas necessidades, ou modificar o artefato gerado.

Modificar o gerador pode ser uma tarefa custosa. Normalmente os geradores são ferramentas específicas para um determinado domínio de problema, de forma que introduzir modificações sem a perda de compatibilidade com esse domínio exige uma análise demorada. Além disso, deve-se realizar essas mudanças de forma genérica, pensando-se não somente no artefato que se deseja modificar, mas em todos os artefatos que serão gerados a partir de então.

Por outro lado, modificar o produto do gerador é muito mais fácil, pois pode-se trabalhar diretamente no artefato que se deseja modificar. Porém, essas mudanças podem se perder caso o artefato seja re-gerado. Para resolver esse problema, normalmente são utilizadas técnicas associadas ao conceito de *round-trip engineering* (HENRIKSSON; LARSSON, 2003; ANTKIEWICZ; CZARNECKI, 2006; HETTEL; LAWLEY; RAYMOND, 2008). Essas técnicas são baseadas em engenharia reversa (HENRIKSSON; LARSSON, 2003), e visam abstrair as mudanças realizadas diretamente nos artefatos gerados, duplicando-as nas especificações de origem. Dessa forma, as mudanças não se perdem, estando presentes nas próximas gerações de artefatos.

Uma abordagem mais simples consiste em não permitir que certas mudanças sejam realizadas. Um exemplo de ferramenta que utiliza essa abordagem é a plataforma NetBeans: o editor textual de código de interface desta plataforma bloqueia certos trechos de código, associados com características estruturais da interface, de forma que o desenvolvedor só pode inserir essas modificações por meio do editor visual. Essa abordagem, porém, é restrita aos casos onde se pode bloquear modificações sem perder a flexibilidade necessária para se resolver os problemas daquele domínio.

A abordagem generativa também pode ser analisada frente aos quatro conceitos da reutilização descritos por Krueger (1992):

Abstração: conforme já discutido anteriormente, o que está sendo reutilizado é o processo de construção dos artefatos, e não os artefatos em si. A abstração desse processo consiste na definição do formato de entrada do gerador, seja ele uma DSL visual, textual, ou mesmo parâmetros que preenchem um ou mais *templates*. Essa definição corresponde à **parte visível** da abstração, com a qual o desenvolvedor trabalha. A **parte escondida** fica dentro gerador, e é responsável pelos detalhes de implementação;

Seleção: a seleção consiste em escolher um gerador apropriado para o problema em questão. Diferentes geradores correspondem a diferentes processos para se produzir os artefatos do domínio, e pode-se decidir por um deles, dependendo do problema. Neste cenário, a situação ideal seria manter uma biblioteca de geradores de aplicação, e utilizar técnicas de classificação e busca para facilitar a seleção, de forma similar às bibliotecas de componentes reutilizáveis. Porém, essa situação requer a existência de um grande número de geradores para um mesmo domínio de problema. Em 1992, Krueger destacava a escassez de geradores, que além de poucos eram normalmente restritos a um número reduzido de domínios (KRUEGER, 1992). Atualmente, com a evolução das tecnologias necessárias para a programação generativa, principalmente a transformação de software, o número de geradores vem crescendo, conforme demonstram Allilaire et al. (2006), que exploram o gerenciamento de repositórios de modelos e transformações de software, utilizando técnicas similares às utilizadas em repositórios de componentes;

Adaptação: *"a adaptação é a tarefa primária realizada por um desenvolvedor de software utilizando um gerador de aplicações"* (KRUEGER, 1992). Corresponde à tarefa de especificação que produz a entrada requerida pelo gerador, normalmente um programa em uma DSL. O desenvolvedor utiliza essa DSL para informar ao gerador como especializar os conceitos do domínio e produzir uma solução para o seu problema; e

Integração: a integração não é um problema quando um gerador produz um programa completo (KRUEGER, 1992). Porém, nos casos onde apenas parte dos artefatos é gerada, é necessário que os mesmos sejam integrados com outras partes, sejam estas geradas automaticamente ou manualmente. Para que essa integração seja automática, os geradores precisam estar preparados e cientes do ambiente ao qual os artefatos gerados serão integrados. Desta forma, os desenvolvedores podem trabalhar somente no nível de abstração da entrada do gerador, especificando o que for necessário para que o gerador possa realizar essa integração sozinho. Porém, essa é a situação ideal. Em outros casos, a única solução é modificar manualmente os artefatos gerados de forma a promover sua integração.

Vale a pena ressaltar que os conceitos de programação generativa, principalmente quando se pensa em reutilização de software, muitas vezes se confundem com os conceitos ligados às linguagens específicas de domínio. Czarnecki e Eisenecker (2000), autores de um dos principais livros sobre programação generativa, destacam a importância das DSLs nessa abordagem. Cleaveland (1988) utiliza os termos "Linguagem de quarta geração" e "Linguagem orientada à aplicação" ao invés de DSL, mas os conceitos de geradores de aplicações que ele apresenta correspondem a um compilador DSL (DEURSEN; KLINT; VISSER, 2000).

A proximidade destas duas áreas também se reflete na evolução das tecnologias envolvidas e da pesquisa acadêmica e industrial. O desenvolvimento orientado a modelos, foco deste trabalho de pesquisa, é um exemplo recente que reúne os conceitos de DSL e programação generativa.

Frameworks Orientados a Objetos

Frameworks orientados a objetos ou *frameworks* de componentes estendem as idéias de bibliotecas de subrotinas e de componentes. A principal diferença, e uma das principais características dos *frameworks*, é a "inversão de controle": enquanto com bibliotecas comuns a aplicação é responsável por realizar chamadas aos artefatos sendo reutilizados, um *framework* é responsável por realizar chamadas à aplicação, detendo o fluxo de controle (DEURSEN; KLINT; VISSER, 2000), (JOHNSON, 1997a) apud (BRAGA, 2002).

Estruturalmente, pode-se definir um *framework* como um conjunto de classes que contém o projeto abstrato de soluções para uma família de problemas relacionados (JOHNSON; FOOTE, 1988) apud (BRAGA, 2002). Essas classes encapsulam conhecimento sobre essa família de problemas, ou sobre esse domínio de problema, que pode ser reutilizado da mesma forma com que se reutilizam componentes de software.

Do ponto de vista de seu propósito, pode-se definir um *framework* como o esqueleto de uma aplicação, que pode ser instanciado por um desenvolvedor de aplicações (JOHNSON, 1997b) apud (BRAGA, 2002). Sendo um esqueleto, um *framework* não define somente as classes de forma isolada, mas também as interfaces e conexões entre as mesmas, assim como a estrutura geral da aplicação instanciada. Dessa forma, ao utilizar um *framework*, um desenvolvedor não está reutilizando somente classes ou componentes isoladamente, que precisam ser posteriormente integrados em uma aplicação, mas sim toda a estrutura interna. Essa estrutura também representa parte do conhecimento daquele domínio, e assim pode-se dizer que o nível de reutilização alcançado com um *framework* é maior do que o nível de reutilização alcançado com componentes isolados, representando um avanço significativo em reutilização (KRUEGER, 1992).

A utilização de um *framework* é normalmente feita por meio de seus "pontos variáveis" (*hotspots*), que são os pontos que definem o que é variável em um domínio de aplicação (BUSCHMANN et al., 1996) apud (BRAGA, 2002). Junto com os ganchos (*hook*), que são os pontos do *framework* passíveis de serem adaptados, os "pontos variáveis" são a forma com que o desenvolvedor normalmente instancia sua aplicação.

Sendo uma técnica de reutilização, o uso de *frameworks* compartilha com as demais abordagens apresentas os quatro conceitos básicos da reutilização:

Abstração: a abstração, como na maior parte das abordagens, se resume a representar o conhecimento do domínio abstraindo-se os detalhes de implementação em uma **parte escondida**, e descrevendo os conceitos de mais alto nível na **parte visível**. No caso dos *frameworks*, a **parte visível** corresponde aos pontos que o desenvolvedor utiliza para instanciar a aplicação, no caso, os "pontos variáveis" e os "ganchos". A **parte escondida** se refere ao restante da estrutura, que é normalmente reutilizada sem modificação, também conhecidos como *frozen spots*;

Seleção: a seleção se resume à escolha de um *framework* adequado para a aplicação que se deseja construir. Neste caso, técnicas de classificação e busca podem ser utilizadas para facilitar a seleção;

Adaptação: a adaptação consiste na instanciação da aplicação, em que o desenvolvedor utiliza os "pontos variáveis" e "ganchos" para criar uma aplicação que atenda aos requisitos; e

Integração: a integração normalmente não é um problema, pois um *framework* já é uma aplicação semi-pronta que, uma vez instanciada, pode ser executada diretamente. Porém, pode ser necessário realizar a integração da aplicação com o ambiente, por exemplo, com o sistema operacional ou um banco de dados específico. Neste caso, essa integração será tão simples quanto for a possibilidade de parametrização do *framework*.

Padrões de software

Uma forma bastante conhecida de reutilização são os padrões de software (COPLIEN, 2006). Sejam de análise, de processo ou de projeto, os padrões têm um único objetivo: representar soluções bem sucedidas para problemas recorrentes, de forma que, ao se deparar com uma situação similar à vivida originalmente, um desenvolvedor possa aplicar a mesma solução, obtendo os mesmos resultados que o criador do padrão. Neste sentido, o objeto da reutilização é o conhecimento obtido na solução desse problema recorrente. Os quatro conceitos básicos da reutilização estão também presentes nessa técnica:

Abstração: a abstração ocorre quando as soluções recorrentes são generalizadas e representadas segundo um formato específico. Existem alguns formatos para descrição de padrão que são mais comumente utilizados, destacando-se aquele utilizado em (GAMMA et al., 1995). Normalmente essa descrição contém uma descrição do problema, de forma que um desenvolvedor possa decidir se esse padrão é adequado ao seu problema ou não;

Seleção: a seleção ocorre quando o desenvolvedor procura por um padrão por meio da comparação do problema descrito no padrão e o problema sendo vivenciado por ele no momento. Esse processo é normalmente manual, exigindo uma consulta a catálogos de padrões, tal como o catálogo online *hillside.net*[5];

Adaptação: a adaptação consiste na instanciação do padrão, adaptando-o para a situação atual. Os elementos do padrão são normalmente genéricos, precisando ser renomeados ou mesmo modificados; e

Integração: a integração exige a adaptação do restante do sistema, ou do projeto, para acomodar os elementos inseridos pelo padrão.

Existem basicamente três abordagens para os processos de adaptação e integração de um padrão (FLORIJN; MEIJERS; WINSEN, 1997): a abordagem **top-down**, na qual os elementos do padrão são criados e então integrados ao restante do sistema; a abordagem **bottom-up**, na qual os elementos já existentes no sistema são "transformados" nos elementos do padrão, e a abordagem **híbrida**, que é uma combinação das outras duas, com parte dos elementos do padrão sendo criados a partir do zero e parte sendo adaptada a partir de elementos já existentes no sistema.

Reengenharia de Software

Outra técnica que promove a reutilização de software é a reengenharia de software. Também conhecida como renovação ou recuperação, tem como objetivo principal a reconstrução de sistemas legados para aumentar sua qualidade e manutenibilidade. Porém, sistemas legados normalmente encapsulam um conhecimento que evoluiu durante anos, e que não pode ser desperdiçado. Dessa forma, a reengenharia de software é também uma forma de reutilizar software, fazendo com que esse conhecimento seja reaproveitado. A reengenharia de software não somente recupera as informações de um projeto existente, mas também as reutiliza para alterar ou reconstruir o sistema, adicionando novos requisitos ou introduzindo novas

[5] `http://hillside.net/patterns/onlinepatterncatalog.htm`

tecnologias. As principais atividades da reengenharia de software são: engenharia reversa, seguida por mudanças no sistema (de funcionalidade ou de implementação), e seguida pela engenharia avante. Em outras palavras, "*reengenharia é o processo de se criar uma descrição abstrata do sistema, elaborar mudanças em alto nível de abstração, e então reimplementar o sistema*" (JACOBSON; LINDSTROM, 1991).

Esse processo de engenharia reversa, modificações e engenharia avante, também contempla os quatro conceitos da reutilização de software. A fase de engenharia reversa corresponde ao conceito de abstração, enquanto que as fases de modificações e engenharia avante correspondem aos demais conceitos:

Abstração: a abstração corresponde à engenharia reversa, onde o conhecimento existente no sistema legado é recuperado, e toda informação relevante é organizada de forma a possibilitar sua futura utilização durante a reconstrução;

Seleção: a seleção ocorre quando o desenvolvedor precisa realizar mudanças no sistema, sejam elas de funcionalidade ou implementação. Ele precisa conhecer cada artefato recuperado durante a engenharia reversa, e selecionar aquele onde serão feitas as modificações;

Adaptação: a adaptação corresponde às mudanças sendo efetuadas nos artefatos recuperados. Os mesmos são modificados para atender a novos requisitos ou incorporar novas tecnologias; e

Integração: a integração ocorre de forma gradual. Normalmente, as organizações preferem utilizar processos de reengenharia gradativos, onde apenas um módulo é reconstruído a cada vez. Isso possibilita que o sistema legado possa continuar sendo utilizado durante a reconstrução (SEACORD; PLAKOSH; LEWIS, 2003).

O Quadro 17 resume as principais técnicas para reutilização de software apresentadas neste apêndice, apresentando as idéias-chave de cada técnica, de acordo com os quatro conceitos da reutilização.

	Abstração	**Seleção**	**Adaptação**	**Integração**
Desenvolvimento Baseado em Componentes (DBC)	Encapsulamento e utilização de interfaces bem definidas	Navegação e busca automática	Parametrização ou modificação interna	Acoplamento de interfaces providas com requeridas
Linguagens Específicas de Domínio (DSL)	Análise de um domínio de problema	Consulta à gramática ou manual da linguagem	Parâmetros definidos na linguagem	Automatizada através de um compilador
Programação Generativa	Definição do formato de entrada	Escolha de um gerador apropriado	Produção da entrada do gerador	Não existe quando é gerado um programa completo. Porém, podem ser necessárias modificações, quando é gerada apenas parte de um programa
Frameworks	Representação do domínio, destacando os "pontos variáveis" e "ganchos"	Escolha do *framework* adequado	Instanciação, através dos "pontos variáveis" e "ganchos"	Normalmente não é um problema, pois o *framework* é uma aplicação semi-pronta
Padrões	Generalização de soluções recorrentes	Busca por um padrão para um determinado problema	Adaptação do padrão para a situação	Adaptação do restante do sistema
Reengenharia de software	Engenharia Reversa	Seleção dos artefatos a serem modificados	Mudanças nos artefatos recuperados, para atender a novos requisitos ou novas tecnologias	De forma gradual, o sistema é reconstruído aos poucos, com cada módulo sendo integrado separadamente

Quadro 17: Resumo das principais técnicas relacionadas aos conceitos básicos da reutilização de software

APÊNDICE B – Relação entre a abordagem e modelos de maturidade

Neste apêndice são descritos em detalhes as práticas do modelo de maturidade em reutilização proposto por Garcia et al. (2007, 2008) e do modelo de maturidade em MDD definido pela iniciativa *ModelWare* (MODELWARE, 2006d).

Também discute-se a relação entre esses modelos de maturidade e a abordagem definida nesta tese. Em cada prática é indicado um símbolo, sendo que o símbolo ☑ indica que a prática está presente na abordagem, e o símbolo □ indica que a prática está ausente da abordagem. Uma breve explicação, destacada em negrito, descreve o raciocínio por trás da presença ou ausência da prática na abordagem.

É importante ressaltar que não foi feita uma análise rigorosa de aderência aos modelos de maturidade, analisando-se por exemplo as características principais dos produtos de trabalho e atividades da abordagem e comparando-se com os elementos de processo descritos nos modelos. O foco aqui foi oferecer uma visão mais ampla sobre o escopo e abrangência da abordagem, frente ao que existe na literatura com relação às atividades e práticas relacionadas à reutilização e MDD, além de oferecer uma descrição mais detalhada dos modelos de maturidade.

Modelo de maturidade em reutilização de software (GARCIA et al., 2007, 2008)

- **Nível 1 - Incompleto**: neste nível, apenas o desenvolvimento de software tradicional é realizado. Práticas de reutilização são usadas esporadicamente ou mesmo ignoradas e desencorajadas pela gerência. Reutilização é fruto de esforço individual, e os custos da reutilização são desconhecidos. **Não existem práticas definidas para este nível**;
- **Nível 2 - Básico**: este nível é caracterizado pela utilização básica de artefatos com potencial de reutilização. Engloba algumas atividades básicas orientadas à reutilização, e a implementação do domínio de forma direta, sem uma preocupação com análise e projeto mais voltados à reutilização;

- □ **AP1 - Manutenção de métodos e técnicas básicas de reutilização**: estes devem ser documentados, analisados e mantidos sempre atualizados, através de revisões periódicas envolvendo a equipe e a gerência. **A abordagem não prevê o gerenciamento e manutenção de métodos e técnicas para reutilização**;
- ☑ **AP2 - Planejamento de reutilização**: deve ser realizado de acordo com um procedimento pré-definido e documentado. O planejamento deve incluir análise de riscos e determinação das abordagens de reutilização a serem utilizadas. O ciclo de vida do processo de reutilização deve também ser identificado durante o planejamento. **Uma das atividades iniciais da abordagem consiste no planejamento da reutilização, incluindo possíveis riscos, e a adoção desta abordagem pode ser considerada como uma decisão de planejamento, assim como a definição de seu modelo do ciclo de vida**;
- ☑ **AP3 - Definição dos objetivos da reutilização**: assim como a estratégia adotada, a definição dos objetivos deve ser realizada por um grupo com autoridade e conhecimento para esta tarefa. Os objetivos e estratégia devem ser documentados. Também devem ser definidos, documentados e implantados os indicadores de desempenho a serem utilizados para determinar se os objetivos estão sendo alcançados no processo. **A abordagem contém uma atividade para definição dos objetivos da reutilização**;
- ☑ **AP4 - Implementação do domínio**: visa produzir artefatos de software reutilizáveis para o domínio, através de codificação, testes, documentação e empacotamento destes artefatos. Envolve basicamente a definição das interfaces providas e requeridas dos artefatos reutilizáveis, testes de unidade com os artefatos, documentação e empacotamento desses artefatos. **Essa é justamente uma das fases da abordagem, que visa implementar um domínio utilizando técnicas do MDD**;

- **Nível 3 - Definido**: neste nível o principal foco de engenharia é o suporte à variabilidade. Enquanto no nível 2 a preocupação era aumentar o nível de reutilização de artefatos individuais, aqui o foco é oferecer um suporte global à variabilidade do domínio, principalmente com as práticas de análise e projeto do domínio. Também neste nível introduz-se a preocupação com o processo de desenvolvimento de uma organização, treinamento e a introdução de uma unidade dedicada à reutilização. Preocupações com a qualidade dos artefatos também começam a aparecer neste nível;

  - □ **AP5 - Controle e monitoramento do processo de reutilização**: deve ser realizado através de um conjunto de mecanismos e métricas que determinam o status

das atividades, e um conjunto de políticas e um plano de ações, responsáveis pelo controle do processo. **A abordagem não define nenhum mecanismo ou atividade para controle e monitoramento do processo**;

☑ **AP6 - Integração com o ciclo de vida do software**: são definidas atividades, sub-atividades e produtos de trabalho institucionalizados para a organização, com base em um modelo de referência. Também é importante a definição de um procedimento para a cooperação entre os desenvolvedores e a equipe de reutilização. **Esta prática consiste justamente na definição desta abordagem e seus elementos**;

☑ **AP7 - Análise de domínio**: envolve a seleção e definição do domínio, análise das aplicações existentes para identificação das características do domínio, definição dos requisitos e escopo do domínio, identificação e documentação dos pontos variáveis e comuns, e a identificação e documentação de restrições adicionais entre as características do domínio. **A abordagem possui uma fase específica para a análise de domínio voltada para o MDD**;

☑ **AP8 - Projeto de domínio**: envolve a definição de uma arquitetura de referência para o domínio, que define a maneira com que os sistemas do domínio serão construídos. Os requisitos, incluindo a variabilidade, devem ser mapeados para soluções técnicas. Padrões de projeto devem oferecer suporte à estrutura variável que serve de base para os aplicativos do domínio. O arquiteto deve reduzir a complexidade do projeto através de abstrações que consigam separar os principais aspectos do domínio. O arquiteto deve modelar as abstrações de forma a permitir o raciocínio sobre elas. Esta área também envolve a avaliação arquitetural, visando detectar falhas e erros antes da implementação começar. **A abordagem engloba as práticas relacionadas ao projeto de domínio, incluindo projeto arquitetural e suporte à variabilidade**;

□ **AP9 - Treinamento em reutilização**: envolve um programa organizacional de treinamento com suporte adequado, um planejamento do treinamento, e o estabelecimento de um comitê de treinamento interno. **A abordagem não prevê atividades para treinamento**;

□ **AP10 - Gerenciamento da unidade de reutilização**: inclui principalmente o estabelecimento de um comitê de reutilização, com um líder, e suporte e financiamentos providos por uma alta gerência comprometida. Este comitê deve possuir regras e responsabilidades bem definidas, ser composto de membros bem treinados e motivados, e possuir canais de comunicação com a organização. **A**

abordagem não define uma unidade dedicada à reutilização;

- □ **AP11 - Programa de métricas**: definição de políticas e objetivos para medição da reutilização em toda organização. Um plano de medição de reutilização deve ser desenvolvido, com mecanismos para coleta, análise e aplicação de dados. Planos de ações para melhoria de processo com base nos resultados das medições também são importantes. **A abordagem não define nenhum tipo de métrica de controle de andamento do processo, ou controle de qualidade**;

- □ **AP12 - Programa de avaliação organizacional**: envolve a definição, por parte da alta gerência, de políticas de avaliação, além do suporte e integração do processo de avaliação na cultura organizacional. O comitê de reutilização, possivelmente junto com o grupo de controle de qualidade da organização, devem desenvolver e documentar os objetivos, planos e procedimentos de avaliação durante o ciclo de vida. Finalmente, o pessoal envolvido deve ser apropriadamente treinado nas políticas, práticas e procedimentos de avaliação. **A abordagem não define nenhuma atividade com este objetivo**;

- □ **AP13 - Avaliação da qualidade dos artefatos**: envolve a definição de um modelo de qualidade, que define políticas, objetivos e atributos relacionados à qualidade dos artefatos reutilizáveis, assim como um processo de avaliação dos mesmos. **Avaliação de qualidade não faz parte da abordagem**;

- □ **AP14 - Engenharia de Aplicações**: engloba as práticas de desenvolvimento com a reutilização de artefatos previamente construídos. Envolve a busca, seleção, adaptação e integração dos artefatos reutilizáveis. Assume-se que estes artefatos tenham sido previamente testados, e portanto nesta área estão somente as práticas de testes de integração e testes de sistema. Também envolve a estimativa de esforço de reutilização, uma vez que pode ser mais vantajoso construir um artefato do zero do que reutilizar um artefato que exige grande esforço de adaptação. **O desenvolvimento com reutilização está fora do escopo desta pesquisa, e portanto a abordagem não possui nenhuma atividade relacionada a esta prática**;

- **Nível 4 - Sistemático**: este é o nível mais avançado que uma organização pode chegar em termos de reutilização de software. Neste nível, o processo está em constante otimização, de acordo com resultados de projetos anteriores. Também aqui a qualidade dos artefatos reutilizáveis é sujeita a um processo mais rigoroso de controle de qualidade. Outras práticas interessantes deste nível 4 incluem a tentativa de se prever e suprir

necessidades futuras em termos de artefatos reutilizáveis, e uma análise de mercado para se avaliaram questões de investimento em reutilização;

- □ **AP15 - Otimização do processo de reutilização**: envolve a identificação das práticas que precisam ser melhoradas, seguida da implementação das melhorias, e medição do progresso de melhoria. A contínua avaliação de novas ferramentas e tecnologias relacionadas à reutilização também precisa ser realizada. **A abordagem não prevê atividades para melhoria e evolução do processo**;
- □ **AP16 - Controle de qualidade dos artefatos**: é um passo além da avaliação da qualidade, com objetivos específicos para os produtos de software sendo incorporados no planejamento da reutilização. O comitê de reutilização também precisa ser treinado em métodos estatísticos para poder melhor avaliar os artefatos frente a seus modelos de qualidade. **A abordagem não trata do controle de qualidade dos artefatos reutilizáveis**;
- □ **AP17 - Previsibilidade dos artefatos reutilizáveis**: consiste na identificação de necessidades futuras em termos de artefatos potencialmente reutilizáveis, visando reduzir o tempo de desenvolvimento em projetos envolvendo estes artefatos. **Não há nenhuma atividade com este objetivo**;
- □ **AP18 - Análise de condições de mercado e retorno de investimento**: busca analisar o nível de retorno dos investimentos em reutilização, de acordo com as condições do mercado e oportunidades para novos negócios. Também envolve a aquisição de artefatos do domínio, caso necessário. **A análise de mercado realizada é superficial, e se resume a uma pesquisa com base em conhecimento dos especialistas, sem atividades específicas para determinação de custos e retorno de investimento**.

Modelo de maturidade em MDD (MODELWARE, 2006d)

- **Nível 1 - Modelagem *ad hoc***: neste nível, apenas o desenvolvimento tradicional é realizado. Práticas de modelagem são utilizadas apenas esporadicamente ou nunca. **Não existem práticas definidas para este nível**;
- **Nível 2 - MDD Básico**: este nível é caracterizado pela utilização básica de modelos, cobrindo atividades simples do MDD. Modelos são utilizados apenas para guiar a implementação e documentação. Tipicamente, apenas modelos técnicos são utilizados, incluindo todos os aspectos de um sistema, sem distinção entre aspectos de negócio ou aspectos específicos de plataforma;

☑ **ENG1 - Decidir sobre convenções de modelagem**: consiste na identificação e seleção das convenções de modelagem a ser utilizadas pela equipe de desenvolvimento. Para isso, é definido o escopo do modelo: quais requisitos podem/devem ser modelados. Também envolve a decisão sobre quais partes do software serão feitas à mão, e quais serão automaticamente geradas. São definidos quais tipos de diagramas serão construídos e utilizados e com quais técnicas de modelagem. **A abordagem possui uma atividade na qual são analisadas as possibilidades de linguagens a serem utilizadas na modelagem**;

☑ **PJM1 - Decidir sobre ferramentas de modelagem**: envolve a seleção de ferramentas que possam satisfazer às necessidades do projeto, de geração de código e documentação. Deve levar em conta objetivos e restrições do projeto, tais como custos e duração. **Assim como na atividade acima ENG1, a abordagem também busca analisar ferramentas existentes para serem utilizadas**;

☑ **ENG2 - Desenvolver modelo técnico**: o modelo técnico descreve os principais módulos do software, incluindo elementos independentes e específicos de plataforma. O modelo técnico representa a divisão em camadas, a comunicação entre as camadas, os componentes a serem utilizados ou desenvolvidos, as interfaces entre os componentes, a plataforma destino e persistência, entre outros aspectos de negócio e técnicos do sistema. **A abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver modelos técnicos**;

☑ **ENG3 - Gerar código a partir do modelo técnico**: ferramentas simples de geração automática de código produzem código que corresponde ao modelo técnico. A saída desta atividade é o esqueleto do produto de software. Envolve também a separação entre código gerado e código não-gerado, e a complementação com código manual para satisfazer aos requisitos, caso a geração não produza todo o código necessário. **A geração de código é um dos principais itens da abordagem, e portanto esta prática está plenamente implementada**;

□ **ENG4 - Gerar documentação a partir do modelo técnico:** similar à geração de código, a documentação de projeto do sistema pode ser gerada a partir do modelo técnico, ou pelo menos parcialmente. Envolve a geração da documentação e a complementação manual, caso a documentação gerada não seja completa. **Apesar de ser possível gerar qualquer tipo de artefato, a abordagem não define atividades específicas para geração de documentação**;

- **Nível 3 - MDD Inicial**: neste nível introduz-se uma separação entre modelos de negócio

(independentes de plataforma) e específicos de plataforma. O objetivo é manter os aspectos de implementação independentes dos aspectos de negócio, de modo a melhorar a eficiência do processo de desenvolvimento. Neste nível, as práticas e artefatos do MDD são institucionalizadas, e um processo definido para o MDD é utilizado pela organização;

- ☑ **ENG5 - Desenvolver modelo independente de plataforma**: um modelo independente de plataforma reflete os requisitos do sistema sem utilizar conceitos específicos da plataforma. Os requisitos são documentados de acordo com as técnicas de modelagem estabelecidas na organização. **A abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver modelos independentes de plataforma**;
- ☑ **ENG6 - Desenvolver transformações técnicas modelo-para-texto**: as transformações devem ser desenvolvidas com base nos metamodelos de origem e destino. A definição das transformações com base na sintaxe concreta (por exemplo, XMI) são insuficientes pois elas limitam a transformação a apenas uma sintaxe concreta. Para isso, decide-se sobre qual linguagem de transformação utilizar, são identificados os metamodelos de origem e destino, são definidas as regras de transformação, que são compiladas e executadas por uma ferramenta, e opcionalmente, completa-se o código gerado, caso necessário, para completar a transformação. **A abordagem tem atividades para transformações de modelo-para-texto, visando dar suporte à variabilidade do domínio**;
- □ **ENG7 - Verificar modelos**: a verificação de modelos inclui a checagem dos requisitos, para determinar se estão completamente e adequadamente refletidos nos modelos. **A abordagem não define atividades para verificação de modelos**;
- ☑ **PJM2 - Modelar o processo de MDD do projeto**: normalmente, cada organização tem um processo de desenvolvimento implantado, que é modificado ou adaptado para atender às necessidades de cada projeto. No caso do MDD, essa adaptação deve considerar que: (i) os requisitos devem ser modelados nas ferramentas selecionadas para a organização, e utilizando os padrões de modelagem pré-estabelecidos; (ii) o foco das tarefas do projeto deve se concentrado nas atividades de modelagem e na definição de transformações entre modelos; e (iii) a maioria dos resultados as atividades de engenharia são modelos e transformações entre modelos. **A abordagem está completamente modelada, e portanto pode ser utilizada como modelo do processo**;
- □ **PJM3 - Aplicar o processo estabelecido**: consiste na utilização do processo definido, e obtenção de métricas para controle de sua execução. **Uma organização**

pode aplicar a abordagem, mas não foram definidas métricas para gerenciar e controlar sua execução, e portanto esta prática não está completamente satisfeita;

□ **PJM4 - Definir, coletar e analisar métricas do projeto**: as métricas devem estar ligadas aos objetivos do projeto. O foco deve ser nas atividades específicas do MDD, como qualidade arquitetural, corretude dos modelos, etc. **A abordagem não define nenhuma atividade referente à definição, coleta e análise de métricas**;

□ **SUP1 - Estabelecer e manter repositórios para modelos e transformações**: com o objetivo de promover a reutilização dos modelos e transformações na organização, repositórios devem ser desenvolvidos e mantidos, de forma que modelos e transformações produzidos em um projeto podem ser reaproveitados em outros projetos. **A abordagem não define nenhum tipo de repositório para o MDD**;

□ **SUP2 - Definir técnicas padronizadas de modelagem e critérios de qualidade**: envolve a definição de um padrão de modelagem para a organização, que representa a evolução das convenções de modelagem definidas no nível 2. Também envolve a definição de critérios de qualidade de modelos, como completude de modelo, consistência inter-diagramas, legibilidade, etc. **A abordagem apenas contém atividades, passos e diretrizes, sem definir técnicas padronizadas ou critérios de qualidade**;

□ **SUP3 - Checar aplicação das práticas**: consiste em verificar se as práticas de modelagem e artefatos produzidos estão de acordo com as técnicas de modelagem e critérios de qualidade pré-estabelecidos. **A abordagem não define maneiras de controle sobre a aplicação de suas práticas**;

□ **SUP4 - Definir métricas padronizadas e procedimentos de coleta e análise de dados**: envolve a definição de métricas para as atividades de modelagem, para os modelos, como coletar as métricas e como analisar os dados. **A abordagem não define nenhum tipo de métricas para controle**;

- **Nível 4 - MDD Integrado**: o nível 4 é caracterizado por uma melhor integração entre os níveis de abstração de modelagem. Aspectos de negócio, independentes de plataforma e específicos de plataforma são separados em elementos do MDD, atividades de modelagem são integradas, e garante-se um desempenho de modelagem eficiente;

  ☑ **ENG8 - Desenvolver metamodelo centrado na arquitetura**: envolve a identificação das principais entidades que fazem parte da arquitetura, os

relacionamentos entre estas entidades, e a linguagem de metamodelagem a ser utilizada. Por fim, o metamodelo é desenvolvido em uma ferramenta que dê suporte à linguagem estabelecida. **O projeto arquitetural voltado ao MDD está presente na abordagem, que tem atividades para que o metamodelo seja centrado na arquitetura**;

☑ **ENG9 - Desenvolver metamodelo independente de plataforma**: de forma similar ao desenvolvimento do metamodelo centrado na arquitetura, o modelo independente de plataforma é desenvolvido modelando-se as principais entidades do sistema e seus relacionamentos, mas sem referências a uma plataforma específica. **A abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver metamodelos independentes de plataforma**;

☑ **ENG10 - Desenvolver modelo de negócios**: consiste na análise dos requisitos de negócio e criação de um modelo que representa como o sistema funciona, utilizando entidades de negócio e relacionamento entre elas. **A abordagem não está focada em um tipo específico de modelo, e portanto pode ser utilizada para desenvolver modelos de negócio**;

☑ **ENG11 - Desenvolver transformações modelo-para-modelo**: consiste na identificação dos metamodelos de origem e destino, padrões de transformação entre os modelos de origem e destino, definição da linguagem de transformações a ser utilizada, e implementação da transformação. A sintaxe concreta deve ser ignorada neste caso. **As transformações modelo-para-modelo estão previstas na abordagem, como uma forma de organização e estruturação dos geradores de código**;

□ **ENG12 - Manter rastreabilidade entre modelos**: significa manter ligações ou mapeamentos explícitos entre modelos, após o resultado de uma transformação. Envolve a definição de um *framework* de rastreabilidade, com componentes e modelos a serem rastreados, tipos de ligações, direção, etc. Os requisitos de rastreabilidade devem ser integrados ao processo de modelagem, que também é responsável por realizar a gerência de configuração e controle de versões das ligações de rastreabilidade. **A abordagem prevê rastreabilidade entre artefatos de análise, projeto e implementação, mas não inclui o rastreamento entre elementos antes e após uma transformação, e portanto esta prática não está completamente satisfeita**;

☑ **ENG13 - Integrar desenvolvimento de produto com desenvolvimento de infraestrutura para uma família de produtos**: esta prática só é aplicável nos

casos onde o desenvolvimento de uma família de produtos é realizado. O objetivo desta prática é separar os modelos técnicos dos produtos daqueles da infraestrutura da família, de forma que ambos sejam mais reutilizáveis. **Esta é a preocupação central da abordagem**;

□ **ENG14 - Simular modelos**: o objetivo desta prática avançada de controle de qualidade é garantir a qualidade dos elementos do MDD o mais rápido possível no ciclo de vida. Simulação de modelos permite a verificação do comportamento modelado de um sistema. Requer a existência de uma especificação formal das ações, em uma linguagem semântica de ações. **Não existem atividades específicas para simulação de modelos**;

□ **SUP5 - Estabelecer e manter limites de desempenho de modelagem da organização**: a organização deve definir e manter os objetivos das atividades de modelagem, e elementos de MDD utilizados, de acordo com cada tipo particular de projeto. Esta prática visa atender à necessidade de alinhamento dos objetivos de negócio da organização com os objetivos do projeto. **A abordagem não tem nenhuma atividade similar a esta prática, que exige um conhecimento sobre diferentes opções de processo para cada projeto, cada um com um grau específico de automação, visando dosar o nível de automação em cada projeto**;

□ **PJM5 - Modelar os processos automáticos de MDD do projeto**: esta prática estende a prática do nível 3, de modelagem do processo, introduzindo limites de desempenho de modelagem, estabelecidos pela organização quando desenvolvendo o modelo do processo. **A modelagem da abordagem se limita à descrição das atividades, não contemplando todos os requisitos desta prática, que exige um conhecimento sobre diferentes opções de processo para cada projeto, cada um com um grau específico de automação, visando dosar o nível de automação em cada projeto**;

- **Nível 5 - MDD Definitivo**: neste último e derradeiro nível, todo o conhecimento da organização é mantido na forma de modelos e transformações, que são o foco do processo de desenvolvimento. Práticas de engenharia de domínio e linguagens específicas de domínio são criadas e continuamente atualizadas;

  ☑ **ENG15 - Desenvolver linguagens específicas de domínio**: linguagens específicas de domínio capturam o conhecimento do domínio utilizando abstrações do domínio, permitindo que os desenvolvedores criem produtos com base em termos familiares do domínio. Consiste na identificação dos principais conceitos

do domínio, definição do glossário do domínio, seleção da linguagem e definição da DSL, através de uma ferramenta. **A abordagem possui uma atividade específica para o desenvolvimento de DSLs no contexto da reutilização**;

□ **ENG16 - Verificar e validar produtos com base nos modelos**: a verificação e validação é realizada através de modelos para teste e validação. Estes modelos (e seus metamodelos correspondentes) capturam os requisitos do sistema, e a verificação ocorre diretamente nos modelos. Desta forma, erros são detectados e corrigidos diretamente nos modelos. **Verificação e validação de modelos estão fora do escopo da abordagem**;

□ **PJM6 - Estabelecer e manter artefatos de modelagem de software estratégicos para o MDD**: consiste na identificação dos artefatos de modelagem (modelos, transformações, etc) que são relacionados com a estratégia de MDD organizacional, na priorização destes artefatos com relação à sua importância frente aos objetivos estratégicos, no agrupamento destes artefatos em diferentes categorias e na criação, armazenamento e monitoramento destes artefatos estratégicos. **A abordagem não contém nenhuma atividade neste sentido**;

□ **PJM7 - Promulgar o modelo de processo do projeto**: consiste na criação de uma instância de um processo para um projeto em particular, no sentido de atribuir recursos para os papéis das tarefas do projeto, estabelecer a duração das tarefas, etc. As decisões são feitas pelo gerente de projeto, seguindo diretrizes organizacionais. Ferramentas de modelagem e suporte também podem ser integradas em um ambiente de execução capaz de orquestrar a execução da instância do processo. **O suporte à execução do projeto limita-se à descrição da abordagem, em termos de suas atividades, papéis, entradas, saídas, etc. Não existe um controle maior sobre sua execução, controle e melhoria, conforme previsto nesta prática**.

APÊNDICE C – Reprodução na íntegra da entrevista referente ao estudo empírico envolvendo o domínio de eventos científicos

A seguir é apresentada, na íntegra, a entrevista realizada como parte do estudo empírico envolvendo o domínio de eventos científicos:

P:O modelo de *features* ajudou na definição das linguagens específicas de domínio, transformações e geradores de código?

R:**A equipe reportou que algumas *features* foram utilizadas diretamente na definição inicial do modelo de configuração, pois descreviam diferentes opções de configuração automática, como quais subsistemas devem estar presentes, e algumas de suas características. Mas principalmente, foi relatado que o modelo de *features* ajudou na obtenção de uma visão geral do domínio, útil para as atividades de customização e vendas, uma vez que permite que o cliente visualize as características do produto de forma facilitada e possa escolher entre as opções de maneira mais rápida. A equipe, que desconhecia a notação empregada nesse modelo, considerou fácil o aprendizado e entendimento, inclusive podendo ser usada comercialmente ou estendida futuramente.**

P:A descrição da variabilidade em cenários (casos de mudança) facilitou a definição das linguagens específicas de domínio, transformações e geradores de código?

R:**Os participantes responderam que estes artefatos ajudaram principalmente na formalização de algumas variabilidades que eram compreendidas de forma implícita pela equipe. Através dos casos de uso e de mudança, as diferentes opções de comportamento ficaram mais claras, porém a equipe relatou que não conseguiu**

identificar uma relação direta entre os casos de mudança e um modelo ou um gerador, como aconteceu para o modelo de *features*. A equipe ficou um tanto surpresa em notar tantas modificações que eram feitas em seus produtos (de forma *ad hoc*) que foram desenvolvidos para atender o domínio científico, mas cada cliente exige mudanças próprias e usando diferentes funcionalidades, sendo que todas essas alterações têm que ser controladas e todo auxílio nesse sentido é importante, seja gerencialmente (pelos coordenadores da equipe) ou tecnicamente (pelos responsáveis pelos casos de mudança.

P:A identificação de candidatos a subdomínio facilitou a identificação das áreas do domínio a serem automatizadas?

R:**Sim, os participantes enfatizaram que conseguiram identificar pelo menos oito subdomínios onde a automação iria ajudar em suas tarefas: configuração dos arquivos de código-fonte (incluindo instalação), banco de dados, relatórios, geração de crachás, geração de certificados, configuração de boleto bancário, configuração de idiomas suportados (incluindo idioma padrão) e processamento de arquivos de pagamento do banco (retornos financeiros). Estes subdomínios consistem principalmente de pontos particularmente problemáticos e trabalhosos para configuração manual, já conhecidos pela equipe após sua experiência com atendimento a seus clientes. Segundo a equipe, o uso do modelo de *features* facilitou na identificação destes pontos.**

P:A identificação de candidatos a subdomínio facilitou a definição das linguagens específicas de domínio, transformações e geradores de código?

R:**Com relação a esta questão, a equipe salientou que a divisão em subdomínios facilitou na definição do escopo dos artefatos de geração de código, que podem ser focados em partes pequenas do problema, tornando seu desenvolvimento mais simples. Porém, eles apontaram a falha de que a existência de múltiplos subdomínios exige também múltiplas etapas na geração de código, já que cada gerador precisa ser executado separadamente para cada subdomínio, o que dificulta quando é necessário gerar código repetidas vezes.**

P:O processo investigativo baseado em decisões para inclusão/exclusão de subdomínios foi utilizado? Se sim, ele facilitou o processo de desenvolvimento dos artefatos do MDD?

R:**A equipe não soube responder esta questão de forma apropriada, pois reportou que apenas foi feita uma decisão inicial de investigação de dois subdomínios, sem que os**

demais pudessem ser investigados.

P:O uso das diretrizes e padrões arquiteturais específicos para reutilização e MDD facilitou o desenvolvimento dos artefatos do MDD e da arquitetura do domínio?

R:**A equipe reportou dificuldades na construção de geradores de código e sua integração ao restante da arquitetura. Porém, eles citaram que os padrões de camada de dados de *features* e a abordagem de visita baseada em *templates* foram muito úteis nesta tarefa. A equipe também reportou que a identificação das diretrizes de variabilidade baseada em *features* de forma isolada para cada subdomínio facilitou o desenvolvimento da camada de dados de *features* utilizada pelos geradores específicos. A unificação de arquivos de configuração para instalação em um modelo inicial que prevê inclusive usuários iniciais para a aplicação foi relatado como grande facilitador para gerenciar a instalação do pacote, que agora pode ser feita por integrantes da equipe menos experientes que verão no modelo reutilizável uma forma de trabalho mais fácil. Antes a instalação dependia de engenheiros experientes no domínio e análise de trechos de código em vários arquivos com busca e alteração dos mesmos (que quando não era feita de forma ideal trazia problemas aos clientes).**

P:A etapa de avaliação arquitetural ajudou a identificar falhas antes de a implementação ser iniciada?

R:**A equipe não chegou a realizar a avaliação arquitetural, argumentando que como o tamanho da equipe era pequeno e não havia a disponibilidade de consultar outros profissionais, considerou que não seria necessário ou produtivo.**

P:As diretrizes de implementação de DSLs, transformações e geradores de código facilitaram a implementação dos artefatos do MDD?

R:**Este foi o ponto mais elogiado pela equipe. Por não terem muito conhecimento sobre o desenvolvimento orientado a modelos, os participantes relataram que as diretrizes de implementação ofereceram dicas importantes na implementação. Em particular, foram citadas as diretrizes para mapeamento das *features* em linguagens, o que segundo a equipe foi um ponto crucial no desenvolvimento dos modelos de configuração da linha. Eles também reportaram que mesmo que nem todas tenham sido usadas neste estudo, elas foram úteis para uma melhor compreensão do potencial desta abordagem e das facilidades que podem ser alcançadas no futuro com o MDD.**

P:O formato de documentação proposto foi adequado, auxiliando na descrição dos artefatos reutilizáveis desenvolvidos?

R:**A equipe não chegou a documentar os artefatos produzidos, alegando que os mesmos foram desenvolvidos apenas como protótipos experimentais, e consideraram mais prioritária a sua conclusão e evolução antes da documentação.**

P:Quais foram as vantagens de se utilizar a abordagem de MDD no desenvolvimento?

R:**Segundo a equipe, a abordagem permitiu atacar problemas recorrentemente enfrentados pela equipe, tais como o alto nível de retrabalho procurando códigos e testando cada mudança no domínio, a existência de arquivos que nunca são utilizados mas que são incluídos nos produtos por conveniência, o que acaba por confundir os desenvolvedores e causando problemas de manutenção, e a necessidade de busca por *links* que precisam removidos dependendo da configuração de produtos adquiridos pelo cliente. A automação conseguida com o MDD permitiu reduzir estes problemas de forma que não vinha sendo conseguida, por falta de tempo e dificuldades em desenvolver uma solução que atendesse a múltiplos clientes ao mesmo tempo. Por exemplo, a equipe citou alguns chamados recentes enviados por clientes via *helpdesk*, referentes à alteração urgente de dados dos certificados emitidos nos eventos, por não estarem de acordo com o exigido ou desejado. Sem o MDD, era necessário criar novo código toda vez que um chamado deste tipo era feito. Após este projeto, que envolveu a implementação do subdomínios de certificados, este trabalho é feito diretamente no modelo.**

P:Quais foram as desvantagens de se utilizar a abordagem de MDD no desenvolvimento?

R:**A equipe citou principalmente as dificuldades de implementação dos geradores de código, pois os mesmos "misturam" código PHP (dos produtos) com código JAVA e marcações do JET, em um mesmo arquivo. Porém, os participantes acreditam se tratar de uma limitação imposta pela dificuldade no aprendizado destas tecnologias**

P:Quais foram as dificuldades para o aprendizado da abordagem?

R:**A equipe relatou dificuldades no aprendizado das tecnologias de modelagem e geração de código, porém com relação à abordagem citaram apenas que tiveram dificuldades em compreender as funções específicas dos casos de mudança na abordagem. Segundo eles, não ficou clara a necessidade de se criar múltiplos cenários para descrever pequenas partes do problema.**

P:Quais foram as dificuldades para definição do modelo de *features*?

R:**A equipe teve dificuldades na identificação correta das *features*, sendo necessária a presença constante do autor desta tese para orientar e corrigir as identificações equivocadas. No geral, os participantes compreenderam os conceitos, mas tinham dificuldade em reproduzi-los no domínio em questão, inserindo constantemente e de maneira equivocada, módulos e funções como sendo *features*.**

P:Quais foram as dificuldades para descrição da variabilidade em cenários (casos de mudança)?

R:**A equipe identificou que a principal dificuldade foi a necessidade de se descrever de forma exaustiva diferentes opções de variantes, em cenários que pouco serviram para o desenvolvimento dos demais artefatos. Segundo a equipe, seria mais fácil descrever apenas as variações mais complexas, utilizando texto e referências ao modelo de *features*.**

P:Quais foram as dificuldades para a identificação de candidatos a subdomínio?

R:**Uma vez que o modelo de *features* estava finalizado, a equipe rapidamente identificou candidatos a subdomínio com base em sua experiência de atendimento a chamados dos clientes. Essa base histórica de uso dos sistemas, incluindo experiências e chamados técnicos registrados com detalhes de interação com cliente, foi importante para essa identificação. Assim, não foram relatadas dificuldades nesta etapa.**

P:Quais foram as dificuldades para realizar o processo investigativo baseado em decisões para inclusão/exclusão de subdomínios?

R:**A equipe não chegou a realizar mais de uma iteração neste processo, tendo optado por incluir dois subdomínios em uma única iteração. Portanto, não soube responder esta questão de forma adequada.**

P:Cite outras dificuldades percebidas durante a utilização da abordagem de MDD desde o início do desenvolvimento.

R:**A equipe também citou as dificuldades referentes à operação do ambiente de desenvolvimento, uma vez que não existe muita documentação a respeito destas tecnologias.**